Hugo Menino Aguiar
Licenciado em Engenharia Informática

Profiling of Real-World Web Applications

Dissertação para obtenção do Grau de Mestre em Engenharia Informática

Orientador: Prof. Doutor João Costa Seco, UNL-FCT
Co-orientador: Eng. Lúcio Ferrão, OutSystems SA

Presidente: Prof. Doutor Carlos Isaac Piló Viegas Damásio
Arguente: Prof. Doutor Maria Antónia Lopes
Vogal: Prof. Doutor João Costa Seco

Novembro de 2010

Profiling of Real-World Web Applications

To my father Carlos,
my mother Otilia
and my sister Loide.
I owe you more than I can ever say.

Acknowledgements

"One man may hit the mark, another blunder; but heed not these distinctions. Only from the alliance of the one, working with and through the other, are great things born."

— Antoine de Saint-Exupery

This thesis is the culmination of a long path started in the beginning of my B.Sc. Many people contributed either directly or indirectly to this work. I hope I have remembered everyone.

To my advisors, João Costa Seco and Lúcio Ferrão. They provided me guidance and support at key moments in my work while also allowing me to work independently. Their careful review of several versions of this manuscript improved the quality of this dissertation. Without them this work would not have been possible.

To Luís Caires and António Melo, they gave me the opportunity to do this thesis in a collaboration between the *Faculdade de Ciências e Tecnologias* of *Universidade Nova de Lisboa* (FCT-UNL) and the R&D team of *OutSystems*. I also ankowledge FCT-UNL for giving me work conditions and partial financial support.

To all my *OutSystems* colleagues for their insightful comments and discussion. Hugo Veiga, Hélio Dolores and Lúcio Ferrão were key elements during the development of this work. They took time away from their own lives to help me on this thesis.

And because I'm the sum of my life experiences and because there is more in life than work, I would like to thank some people that helped me along time, either shaping my character or sometimes righteously putting me on the wrong track.

Arlindo Lima, in the beginning of my B.Sc, when I was kind of lost, you were a key element helping me in several subject courses.

Raúl Testa, João Paiva, Francisco Valente, Nelson Seabra, Ricardo Figueiredo, Vitor Rodrigues and Pedro Dias, I could not do the things I have done without your loyalty and friendship. Pedro, thank you for proofreading this manuscript.

All the friends I made in Poland, during my erasmus year. You made my life in Poland incredibly enjoyable and sociable. At least these I have to refer, Inês Nolasco, Filipe Mateus, Carlos Tomás, Ludgero Santos, Ely Teixeira, Maria João Rosa, Lukasz Kulbacki, Adam Gluszuk, Pedro Vareda and Diogo Paulino.

All the guys that helped making University an easygoing and straightforward challenge. I shall enumerate just a few, Pedro Afonso, Sofia Gomes, João Gomes, João Sousa, Bruno Teixeira, Marco Teixeira, Nuno Martins, Sofia Canelas, Tiago Amorim, Nelson Fonte and Ricardo Neves. Fabio Santos and João Ferreira, thank you for sharing car trips with me between our hometown and Lisbon.

Associação Fazer Avançar, making time to go on community service projects contributed greatly to retain my sanity. There are more important things out there in the real world than writing thesis.

Finally, I would like to thank my loving Cátia for putting up with me and my laptop for days and nights. Thank you for your faithful support, courage and strength. My sister, thank you for being my buddy and for helping me grow and avoiding mistakes. And my parents, you always give me the freedom and opportunity to pursue my dreams, even when they appear incomprehensible or dangerous.

All errors and limitations remaining in this thesis are mine alone.

Abstract

The increasing dependency of business on web technologies causes a greater need for accurate assessment of factors associated with the success of enterprise web applications. Performance is one of these factors. Nevertheless, performance evaluation is usually only a concern when problems arise as a consequence of bad-user experience.

Before being deployed to production, applications are assessed and analyzed in simulated environments. However, it is not easy to simulate a real-world environment and the effective use of the system, leading to poor and expensive performance data collection. Moreover, in agile methodologies, where development is focused in fast time to market and getting early feedback from end-users, upfront estimation and forward thinking about scalability is not in the top priorities. This constrains even more performance analysis and tests, as developers are only aware of performance issues when critical feedback from production systems is given back to development. All this, commonly leads to enterprise web applications with scalability problems, and low responsiveness.

This dissertation presents a structured way of giving continuous and real world performance feedback to developers of enterprise web applications. By having early access to real performance metrics, developers easily detect stress points in applications, allowing for timely tuning actions, before reaching critical conditions for end-users. Metrics also help developers assessing the impact of changing intensively used parts of existing applications.

In the first part of this thesis we present a structured overview of the state of the art on the subject. Some profiling techniques are studied as a basis for our design and implementation decisions.

We then focus on the solution architecture, we design and implement a system to run in real world *OutSystems* environments. We also describe how our system collects, aggregates, and transports data, and how is it made available to the developer, crossing different architectural layers of the *Agile Platform* and avoiding significant impact.

To finish, we conclude with the validation of the profiling system and with the results of this thesis.

Keywords: Profiling of Web Applications, Profiling in Production Evironments, Agile Development, Measuring Time and Frequency, OutSystems Agile Platform.

Sumário

O mundo empresarial está cada vez mais dependente das aplicações web. Esta dependência requer uma análise exacta sobre os factores associados ao sucesso destas aplicações. A *performance* é um destes factores. No entanto, a *performance* é geralmente analisada apenas depois dos problemas serem expostos por consequência de experiências negativas do utilizador final.

Antes das aplicações serem colocadas em produção, estas são avaliadas e analisadas em ambientes simulados. Contudo, não é fácil simular o mundo real e o uso efectivo do sistema, isso faz com que a recolha de dados de performance seja ineficiente e cara. Além disso, as metodologias ágeis focam o desenvolvimento na rapidez da chegada do produto ao mercado e na antecipação do *feedback* dos utilizadores finais, deixando de parte estimações e preocupações sobre a escabilidade das aplicações. Isto restringe ainda mais a análise de testes de desempenho, pois os programadores só tomam conhecimento destes problemas quando é dado um *feedback* crítico relativo ao comportamento das aplicações. Por todas estas razões, as aplicações têm normalmente problemas de escabilidade e de resposta.

Esta dissertação tem como objectivo, desenhar e implementar um sistema de *profiling* capaz de recolher métricas em ambientes de produção, e disponibilizá-las aos programadores. Este sistema irá fazer com que os programadores possam tomar decisões de optimização de código mais informadas e ajudará a identificar *bottlenecks*.

Na primeira parte deste documento, apresentamos um resumo estruturado sobre o estado actual das investigações feitas sobre o assunto e são estudadas algumas técnicas de *profiling* que serviram de base nas decisões tomadas durante a fase de implementação.

Posteriormente, centramo-nos na arquitectura da solução. É projectado e implementado um sistema para correr em ambientes de produção *OutSystems*. Também descrevemos a forma de recolha, agregação, disponibilização e transporte de dados, atravessando diferentes camadas da arquitectuta da *Agile Platform* - usada aqui como caso de estudo - sem criar impacto significativo.

Para finalizar, concluímos com a validação do sistema de *profiling* e com os resultados desta tese.

Keywords: Profiling de Aplicações Web, Profiling em Ambientes de Produção, Agile Development, Medições de Tempo e Frequência, OutSystems Agile Platform.

Contents

List of Figures

Listings

Introduction

1.1 Motivation

Web based applications are widely used throughout industry, education, government and other institutions. For instance, a study from NetCraft [9] estimates that on June 2010 there were approximately 206 million sites on the Internet.

The increasing dependency of business on web technologies causes a greater need for accurate assessment of factors associated with the success of enterprise web applications. Non-functional requirements like performance is one of these factors. However, performance evaluation is usually only a concern when problems arise as a consequence of bad end-user experience. Only in special cases there is the anticipation of performance issues, by earlier experiences or obvious expectations, and a thorough performance evaluation is performed.

Hence, correctness and performance are usually assessed and analyzed in simulated environments before being released into production. However, it is not easy to simulate real-world environments and the effective use of the system, thus leading to poor and expensive performance data collection and estimation of the whereabouts of application bottlenecks.

Agile methodologies focus in the fast time to market of software development and evolution based on early feedback from end-users. This causes development and maintenance to be quite modular and raises another concern which is the developers' awareness about performance issues when editing other developer's code.

When developers work on a web application with considerable history, they are not usually aware about the usage and responsiveness of the existing system. Since they are not informed

about the performance of the application in production, they commonly introduce features that lead to performance gaps that have to be corrected later. The fact that no runtime application performance data is usually available, means that execution bottlenecks are only detected when end-users give critical (negative) performance feedback.

With our research we want to answer two different questions:

- Can we collect real world data about performance of web applications without significant impact in the end-user experience?
- Can we give feedback to developers in a way that impact analysis is improved and that anticipation of performance issues is achieved?

For that, we present a possible solution of a profiling system to run in real world *OutSystems* environments. By measuring realtime performance, the profiling system helps developers to detect stress points in applications, allowing for timely tuning actions before reaching critical conditions for end-users. These metrics also help developers assessing the impact of changing intensively used parts of existing applications.

1.2 Thesis Context

This work was developed while integrated in the Research and Development (R&D) team of *OutSystems*, a software company founded in 2001, with offices in Portugal, Netherlands and United States. The main solution of the company is the *OutSystems Agile Platform*, a tool to develop web applications that evolve over time.

There are two main reasons why it is interesting to study this subject using *OutSystems*. First, because *OutSystems* share the same motivations that were described in *Section* 1.1. And second, because of the interesting characteristics of the *Agile Platform* that seemed to make possible the creation of a profiling system to run within it.

Profiling of web applications is not an easy and common task in regular production environments. There may be some explanations for that fact, among which, we find the heterogeneous context of web applications usually containing interface code, business logic and databases. Collecting data in all tiers of an application and gathering it in a meaningful way is not a trivial task. On the other hand, performance degradation caused by instrumentation and data collection is usually the reason for not collecting real-world data.

The *OutSystems Agile Platform* [10] integrates the development of web applications in one single programming language and one development environment that supports the whole life cycle of applications. By presenting a unified solution, the *Agile Platform* has connections between development and production environments which allows to implement a complete collect, transport and visualization solution. The implementation of this work also benefits from services, already present in the platform, to handle the data transport between the different layers of the architecture.

Web applications are developed using a domain specific language (DSL) [31, 43] that integrates interface design, business logic and database manipulation operations in a single language. Applications are then compiled to standard main stream technologies and applications are set to run on a standard application server architecture. In our proposed solution, by instrumenting the generated code with efficient collecting profiling techniques, probes are placed in the applications. So, when a user interact with a application, these probes are reached and data is recorded and stored.

The high level of abstraction provided by the *OutSystems* programming language, leads developers to be unaware of many implementation details and to focus on the meaning of their programs. It is true that the platform takes care of many code optimizations, but there are some cases where the optimizer is simply not enough and where applications would benefit from a clever design. By collecting simple runtime data at the level of the DSL, as execution counts and duration, and considering as targets its course grain elements, and avoiding low level monitoring we achieve two important milestones, we keep the interference level and the performance impact in production environments at an acceptable level and we produce information that is tightly connected to the programming elements and is easily shown to developers in the development tool.

1.3 Web Applications Life Cycle

In order to describe the *OutSystems* applications life cycle and their target architecture we use two types of environments: the development environment meaning the environment in which developers program, design and test web applications, and the production environment where the application goes out to the world and reaches final end-users. For the sake of simplicity we omit here the quality assurance environment usually used to ensure the quality of the application, open bugs and review bug fixes. We also omit some details of the development and production environments but we will address them on the next chapter, in *Section* 2.4.

Consider the simplified life cycle of web applications depicted in *Figure* 1.1. Development starts in the *Service Studio*, the visual development tool of the *Agile Platform*, that is typically connected to a controlled development environment. The application code is compiled and published in that environment for the first testing phase. Depending on the actual staging architecture, the code is manually transported, by a delivery manager or a gatekeeper person, to a production environment and put in use. In the standard installation, there are ad-hoc processes for collecting users' feedback and getting it back to the development teams.

After the application has been deployed to a production environment and users start to use it, the delivery manager will receive feedback about the users' experience. This feedback is then passed to the developers for making changes in the application.

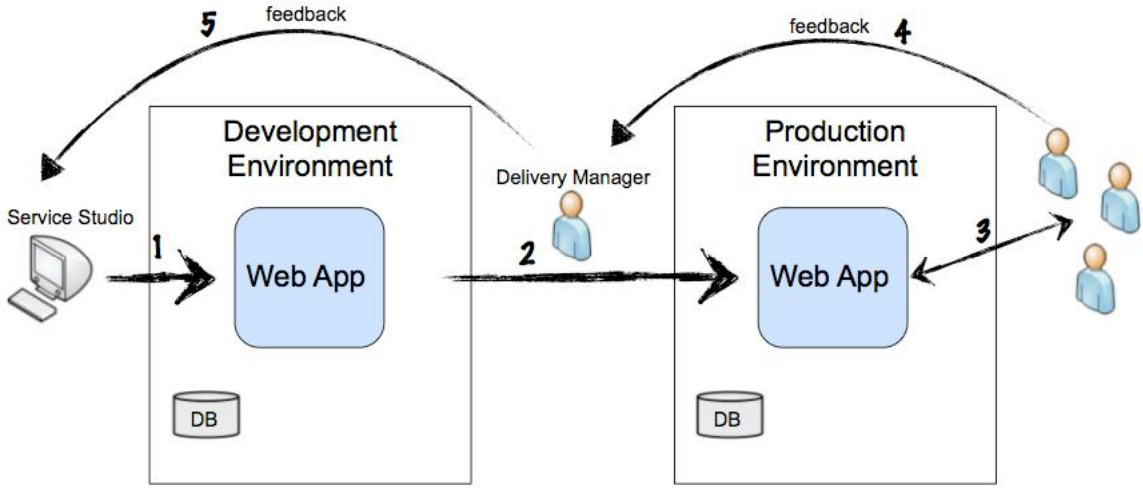

Figure 1.1: Simplified web application life cycle

If we simplify the current web applications life cycle we can summarize it in five main steps (see *Figure* 1.1):

1. Developers use *Service Studio*, to design, create, modify and test web applications. When necessary, applications can be published to the development environment to be tested and analyzed.
2. When applications pass a test phase and there is a decision to deploy it, the delivery manager transports it to the production environment.
3. Users interact with the application.
4. The delivery manager receives feedback about the users experience.
5. The feedback is analyzed and passed to developers for making changes in the application. The cycle restarts in step 2.

1.4 Goals

The goal of this work is to reach the target architecture depicted in *Figure* 1.2. For that, we propose an extension of the existing life cycle, where data is continually collected and transported to the development environment. Note that *Service Studio* always runs connected to an environment and therefore it can retrieve collected data and show it to the developer. In this way, the availability of this performance information, anticipates the need of explicit feedback from users to detect bottlenecks and stress points of the application. We next present the main steps of *Figure* 1.2 in more detail. Although we omit here the feedback given by clients and end-users, we do not intend to replace this explicit feedback but only to anticipate needed changes to the applications.

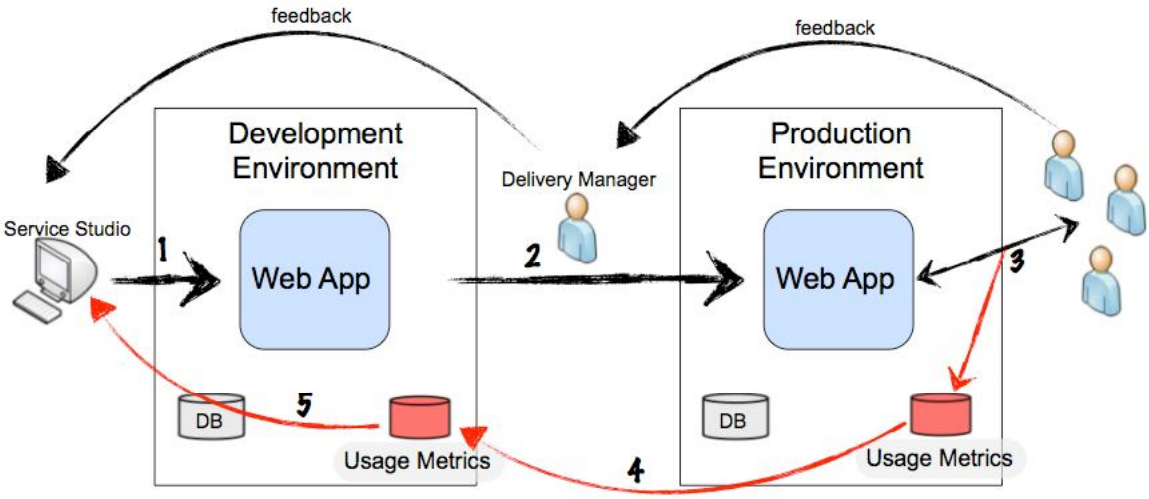

Figure 1.2: Target architecture

1. Developers use *Service Studio*, to design, create, modify and test web applications. When necessary, applications can be published to the development environment to be tested and analyzed.
2. When applications pass a test phase and there is a decision to deploy it, the delivery manager, transports it to the production environment.
3. The application runs for a period of time in the production environment where the profiling system measures and stores relevant metrics.
4. Metrics are transported back to the development environment.
5. Runtime performance data is shown in *Service Studio*. Developer can then visualize information about the performance that each programming element of the application has in production.

The final objective of implementing a profiling system to run in the *Agile Platform* is therefore to collect data efficiently, without causing impact on performance, and showing it to developers in the context where it is most necessary and useful.

1.5 Challenges

Among the goals defined above there are challenges that arise - it is essential to answer the following questions:

1. Which information, concerning web applications analysis, is more relevant to the developer and what can we measure inside a production environment?
2. How to integrate a profiling system inside *OutSystems* production environments without significant impact in application performance?

3. How to decorate *Service Studio* with profiling information without cluttering the existing environment and without significant impact to its performance?

4. How to send data back to the development environment? In the present situation, production and development environments are always typically disconnected.

1.6 Approach

In this document, we describe the design and implementation of a profiling system to analyze the performance of *OutSystems* web applications. This system collects runtime performance data in production environments and automatically transports it to the development environment tool, thus allowing developers to make more informed decisions on code optimizations, anticipating the critical feedback from end-users and helping to identify bottlenecks and stress points that are sometimes difficult to detect.

In the first phase of this work, we inquired some key *OutSystems* developers to find out which metrics are more relevant to measure on a production environment. The results of this inquiry helped us avoiding the risk of implementing a system that wouldn't reach its goals in this real world context. We also studied some techniques that guided our decisions along the design and implementation phase.

During the development phase, we created a system to collect and aggregate metrics in production environments, a system to transport metrics to development environments and we proposed a visualization solution, to show the metrics inside a development tool.

Since we aimed at providing automatic feedback in *Service Studio*, we connected the production and development environments - in the existing architecture they are disconnected.

Finally, our solution is functionally complete, up and running. It is expected to see this profiling system in a future release of the *Agile Platform*, contributing to the quality of development and maintenance of *OutSystems* applications.

During this work we also wrote and submitted a paper, entitled *Profiling of Real-World Web Aplications* [13], to the International Symposium on Software Testing and Analysis, Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging, where it was later presented (13th July in Trento, Italy).

1.7 Document Structure

In *Chapter* 1 we give a vision about the context of this thesis. And we also present the current life cycle of the *OutSystems* web applications, the goals and the challenges of our research.

We next describe the architecture of the *OutSystems Agile Platform* giving more importance to the components that are more relevant for this work. We focus first on the development tool, the *Service Studio*, and the main language constructions. We then describe the runtime support system, the *Platform Server*, and the process of deployment of web applications (*Chapter* 2).

Chapter 3 visits the basic notion of profiling and presents the inquiry made to experienced *OutSystems* developers, that helped us understanding which properties are more relevant to be measured. Then we describe some techniques and methodologies for profiling (*Chapter* 4).

In *Chapter* 5 we describe the most relevant aspects of the work developed. We show how our profiling system collects, aggregates and transports data from the production environment to the development environment crossing different architectural layers of the *Agile Platform* avoiding significant impact.

Finally, the results analysis is addressed in *Chapter* 6 and the conclusions are presented in *Chapter* 7 where we also describe the conditions and the different phases of the project development.

Agile Platform

The *OutSystems Agile Platform* [10] is composed by several heterogeneous parts that contribute to integrate the development, staging and execution of web applications. In this chapter, we focus first on the development tool of the *Agile Platform*, the *Service Studio*, and the *OutSystems* programming language. We then describe the runtime support system, the *Platform Server*, which includes a *Database Server*, several *Front-end Servers* for load-balancing purposes, and a *Deployment Controller Server*.

In particular, we describe the inner components of each *Front-end Server*. We also explain the deployment process of *OutSystems* web applications and present the implementation of an environment.

2.1 Service Studio

Service Studio is the development environment of the *OutSystems Agile Platform*. It allows a developer to design a complete web application in a single environment. Web page interfaces, business logic, database tables and security settings are all set in this single and integrated environment. The language of *Service Studio* is graphically oriented, all elements are visually defined by dragging and dropping smaller elements and defining specific properties. Applications created using *Service Studio* can be compiled and published to the *Platform Server* and accessed via web browsers. The *Platform Server* is the runtime support system for *OutSystems* web applications, we give more details about this component in *Section* 2.3.

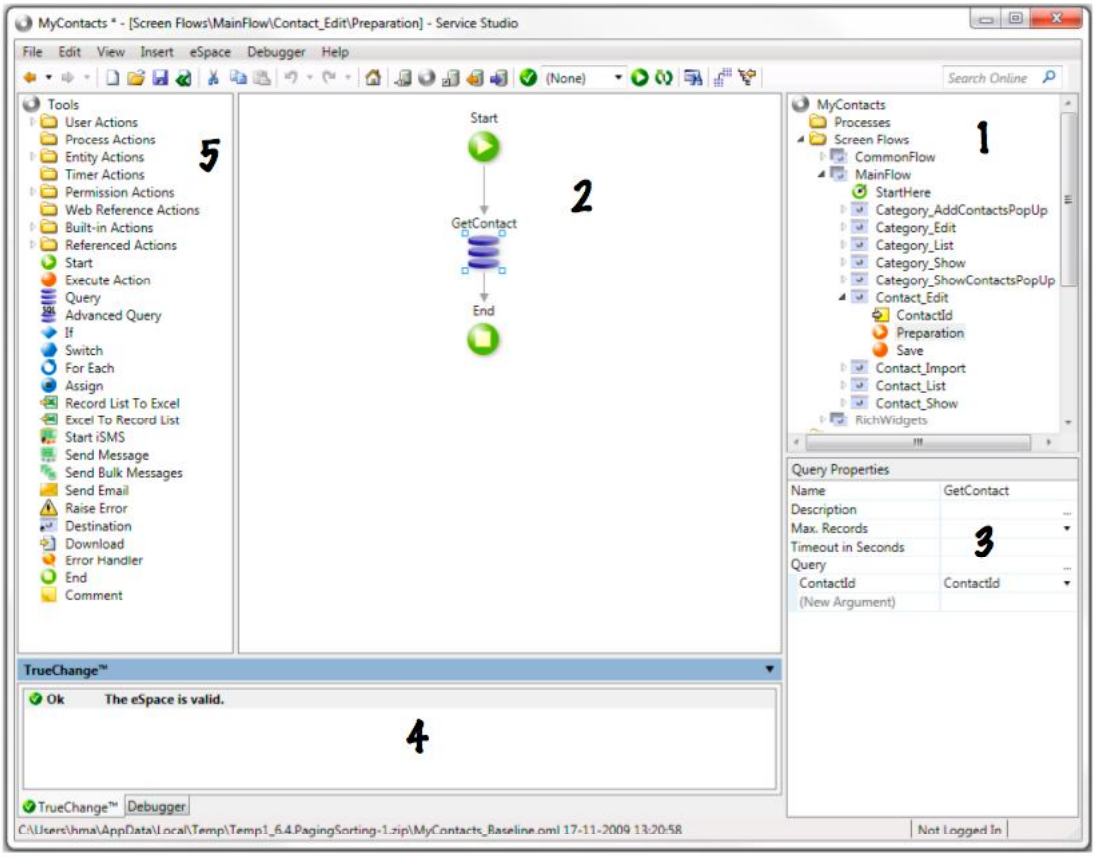

Figure 2.1: *Service Studio* layout

The layout of *Service Studio* interface is depicted in *Figure* 2.1 and contains the following elements:

1. The *eSpace tree* shows all the elements available in the *eSpace*.

2. The *Flow Canvas* where the developer designs the screen or *Action Flows*.

3. The *Properties Pane* where the developer can see and define the properties of the selected element, either in the *Flow Canvas* or in the *eSpace Tree*.

4. The *Lower Pane* contains two tabs: 1) *TrueChange* where the developer can check for *eSpace* errors and warnings; 2) *Debugger* where the developer can observe the runtime behaviour of the *eSpace*.

5. The *Tools Tree* contains the elements that can be added to the flow. For example, the developer can drag conditional nodes, assignments, queries, actions calls, or iteration calls to the flow.

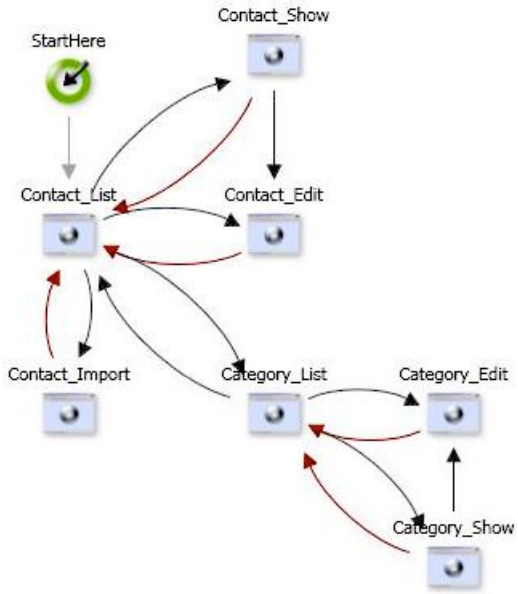

Figure 2.2: Example of a *Web Flow*

2.2 Visual Programming Language

A DSL [31, 43] is a programming language or executable specifcation language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain.

Service Studio implements a domain-specific language designed to represent web applications through high level constructs. With simple constructions, the developing tool interact with diverse components of the system, facilitating the communication with the data repositories, the manipulation of data and the interface with the user. The main high level elements of the language are *Web Flows* which define the connections between web pages, showing the possible end-user interaction sequence, *Web Screens* and *Web Blocks* which graphically define the interface of an application, *Action Flows* that define pieces of logic of an application, and *Entities* that define the data model. All these elements are integrated by the tool with clear benefits to correctness that in most cases is forced by design.

Figure 2.2 depicts the *Web Flow* of a simple web application implementing an address book where the end-user can create, update, remove contacts from a database. A *Web Flow* defines the entry point of an application, in this case it is screen *Contact List*, and which are the next possible screens for each situation. For example, screen *Contact List* may lead to screens *Contact Show*, *Contact Edit*, *Category List*, or *Contact Import*. *Screens* are web pages programmed to interact with the application code in the server.

Action Flows define a piece of behavior that may be triggered by an end-user when interacting with a screen, by following a link and loading a new page or by pressing a button.

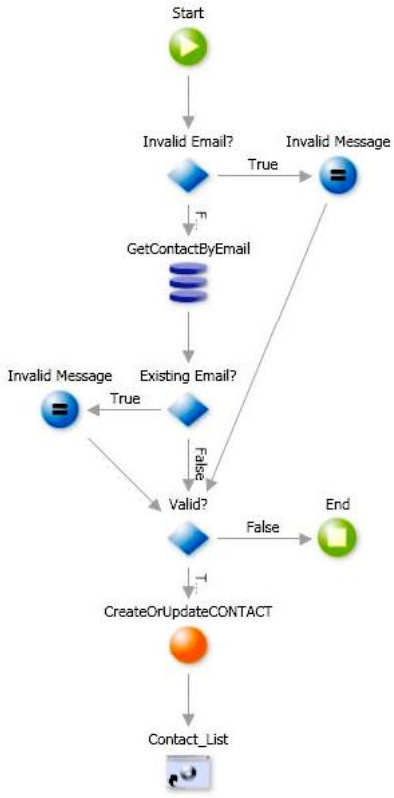

Figure 2.3: Example of an *Action Flow*

Action Flows are visually modeled using basic programming elements, e.g. assignments, queries, conditional and loop constructs. *Figure* 2.3 shows an *Action Flow* that inserts a new contact in our sample database. This *Action Flow* is called when the user presses button "save" in screen *Contact Edit* after filling all necessary information in a form. Notice that action *CreateOrUpdateCONTACT* (predefined by the system) tries to update the entity and if it fails it creates a new one.

Below we briefly describe the main language constructions integrated in *Service Studio*:

- **Start & End** - Delimit the action flow.
- **If & Switch** - Control the execution flow by evaluating expressions.
- **Assign** - Allows to assign a value to a variable.
- **Foreach** - Executes a single or a collection of actions for each element of a list.
- **Simple Query** - Executes a database query. *Service Studio* provides a graphical interface to define the parameters, entities, join condition and sorting.
- **Destination** - Deviates the execution flow to a web page.
- **Execution Action** - Executes a specified action.

Figure 2.4: Example of the *OutSystems* language nodes

For more information about the language constructs and *Service Studio* please refer to *Chapter Designing Actions* in [11].

Now we will present two examples of a run of an application. The first one represents a runtime sequence to display data and is depicted in *Figure* 2.5. The second one represents a runtime sequence to submit data and is depicted in *Figure* 2.6. The numbers shown on the figures are referred on the examples description.

When a user types the address of the web application in a browser, a request is sent to the application server (see *Figure* 2.5).

1. The entry point in the web application determines that it should display the ContactList screen. So the request is made for the ContactList screen.
2. The *Screen Preparation* always runs before the screen is rendered. In this case the *Screen Preparation* is used to get data to display on the screen.
3. The query is performed.
4. The query output is used as the screen data source.
5. The screen is drawn and sent back to the browser.

The layout of the screen is designed using the *Screen Editor*, usually with a kind of a form with inputs for the end-user to type data, and a button to submit the form. To model the behaviour of the button the developer uses a *Screen Action*. A *Screen Action* runs on a specific event on a screen, usually the click of a button (see *Figure* 2.6).

1. A request is made to a screen that displays an empty form.
2. The end-user types data and clicks on the save button.
3. The *Screen Action* that specifies the behaviour of the button is triggered.
4. The database is updated using the forms data.

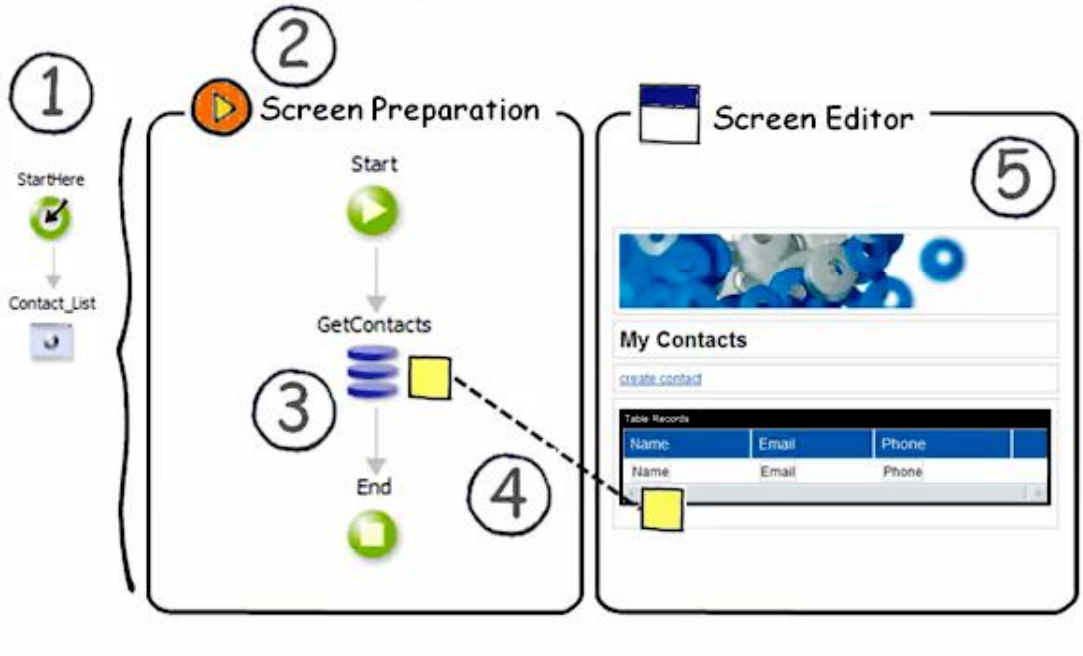

Figure 2.5: Displaying data example

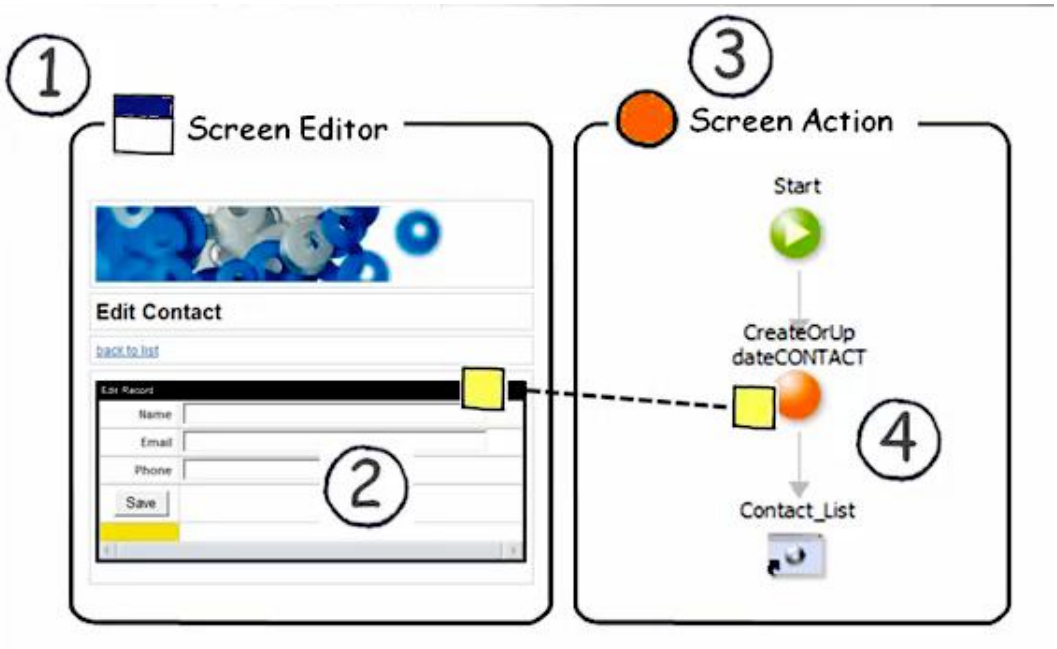

Figure 2.6: Submitting data example

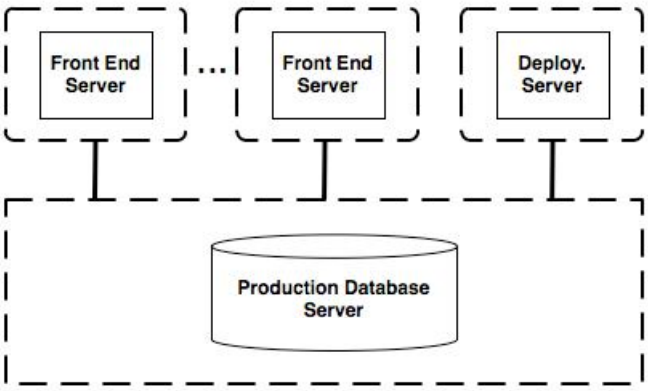

Figure 2.7: Architecture of the *Platform Server* using multiple *Front-end Servers*

2.3 Platform Server

Platform Server is the runtime support system for *OutSystems* web applications. A *Platform Server* may be installed in a farm configuration for scalability and high availability purposes. In this configurations a load balancer distributes web requests among multiple *Front-end Servers*.

The architecture of the *Platform Server* is depicted in *Figure* 2.7 and is composed by: the *Front-end Servers*, the *Deployment Controller Server* and the *Database Server*:

- **Front-end Server**: A *Front-end Server* is a typical web application server with some extra services:
  - *Service Center* - a console to administrate the *Platform Server*; it provides centralized access to all *Platform* resource information as application versioning and management, runtime activity and runtime execution reports.
  - *Log Service* - a service to asynchronously store errors generated by all running applications.
- **Deployment Controller Server**: Is in charge of compiling web application projects, and deploying the compilation result in the *Front-end Servers*.
- **Database Server**: A relational database management system, such as Microsoft SQL Server or Oracle.

Our profiling system must collect performance data in each *Front-end Server*. The way we collect, aggregate the data and transport it will be addressed in *Chapter* 5.

2.4 Process of Deployment

1-Click Publish (1CP) is the process for deployment of a web application into an environment. In *OutSystems* a web application project is known as an *eSpace*. An *eSpace* is edited using *Service Studio* and can be published to a development environment, to be tested and analyzed, or published to a production environment. When the developer invokes the *1CP* process, *Service Studio* contacts *Deployment Controller Server*, which generates the web application code and to deploy it to different *Front-end Servers*.

The environment architecture and the *1-Click Publish* process is depicted in *Figure* 2.8 and *Figure* 2.9. The *1CP* process comprises the following steps:

1. The developer invokes the process *1-Click Publish* that sends the *eSpace* definition to *Deployment Controller Server*.

2. The *Deployment Controller Server* in the web server, receives the *eSpace* and generates a standard .NET or J2EE application code.

3. The *1CP* operation ends with the deploy process - an operation that updates the *eSpace* published version (area that contains the last published version for a specific *eSpace*). In the end, the application is accessible through web browsers.

Both development and production environments contain the ingredients depicted in *Figure* 2.8: a *Deployment Controller Server*, a web server with a database, an application server that runs inside the web server and the running web applications.

The purpose of this chapter is to describe the relevant components of the *Agile Platform* for this work.

We first described *Service Studio*, the development tool for creating web applications, where we want to implement a visualization solution to provide performance metrics in the level of the DSL elements. We then focused on the *OutSystems* language that integrates interface design, business logic and database manipulation. Applications are designed in a single language and are then compiled to standard technologies to run on a server architecture, the *Platform Server*. In this work we want to prepare the compiler with efficient collecting techniques, so during the deployment process, probes will be placed in the applications to collect runtime performance data.

By presenting a unified solution, the *Agile Platform* has connections between development and production environments which allows to implement a complete collect, transport and visualization solution.

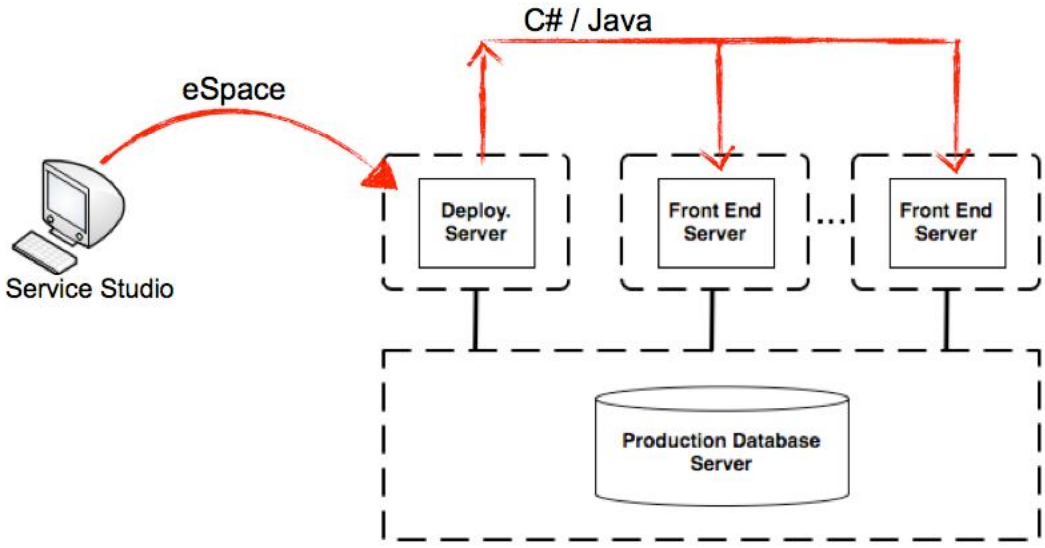

Figure 2.8: Environment implementation and *1-Click Publish*

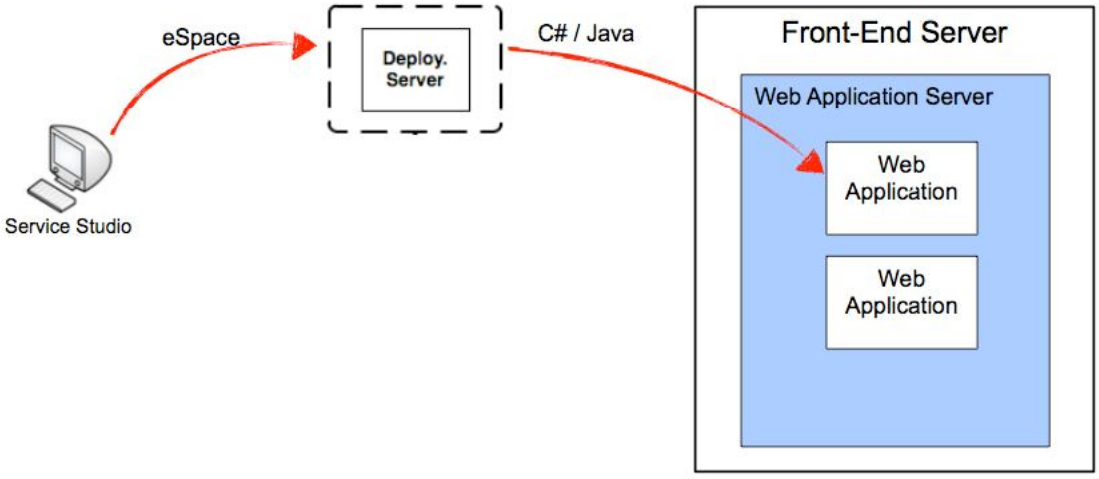

Figure 2.9: Another *1-Click Publish* example

3

Measures and Metrics

This study is about profiling of web applications, we want to measure properties in a real world environment and send the collected data, back to a development environment to be visualized inside *Service Studio*. In this chapter, we define profiling and we present the results of an inquiry made to the most experienced *OutSystems* developers. This inquiry, helped us to be focused on what is really necessary, preventing the risk of implementing a system that wouldn't reach its goals in this real world context.

3.1 Profiling

The importance of profiling has been emphasized in areas as science, management and engineering for many years - profiling in is clear sense of measuring attributes of known objects. We always needed to *"measure what can be measured, and make measurable what cannot be measured"* (Galileo Galilei's). Probably the best statement about importance of measurement is Lord Kelvin's [1] [1]:*"When you can measure what you are speaking about, and express it in numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind. It may be the beginning of knowledge, but you have scarcely, in your thoughts, advanced to the stage of science."* [19]

In software engineering, program profiling, software profiling or simply profiling, is the investigation of a program's behavior using information gathered as the program executes. Profiling is all type of activities related with the dynamic analysis of code, thus analysis of an application execution. Profiling tools are essentially useful when we want to optimize an

[1] William Thomson (or Lord Kelvin), (26 June 1824 - 17 December 1907) was a British mathematical physicist and engineer.

application or detect potential problems of resources usage such as memory. The usual goal of this analysis is to determine which sections of a program to optimize, for example, to increase its speed or to decrease its memory requirement.

There are specific types of profilers, depending on what they report, for example count profilers, function profilers, call graph profilers and trace profilers - all these examples are described in [24].

Profilers use a wide variety of techniques to collect data, below are some examples of these techniques.

- **hardware interrupts** - An hardware interrupt causes the processor to save its state of execution and begin execution of an interrupt handler.
- **code instrumentation** - Code instructions that monitor specific components in a system. This technique inserts code into the program to analyze it.
- **instruction set simulation** - Is a simulation model, it mimics the behavior of a mainframe or microprocessor reading instructions and maintaining internal variables which represent the processor's registers.
- **operating system hooks** - Covers a range of techniques used to alter the behavior of an operating system, applications, or other software components by intercepting function calls or messages or events passed between software components. The code that handles such intercepted function calls, events or messages is called a hook.
- **performance counters** - A set of special-purpose registers built into modern microprocessors to store the counts of hardware-related activities within computer systems. Often used to conduct low-level performance analysis or tuning.

It was already written that there are different types of profilers depending of the the information that they report. But profilers are also characterized by the way they proceed [23]. They are divided in two main categories:

- **Statistical profilers** [23] - A sampling profiler does not require instrumentation, it proceeds by a statistical analysis - periodically (regular intervals of time) looks at which code is currently being executed by the application. As it is statistical, it doesn't see all code, it actually sees the code that is taking more time to be executed. These profilers are considered not as intrusive to the target program, and thus don't have as many side effects. They are used to find bottlenecks in production code as they focus on what code is taking more execution time.
- **Instrumenting profilers** [32] - Some profilers instrument the target program with additional instructions to collect the required information. They work by changing an application source code or binary, and adding calls to functions that count how many times each procedure was called or how much time was spent inside. This approach allows for an exhaustive analysis. However, instrumenting always has some impact on the program

execution [41]. It can be minimal depending on the placement and the mechanism used to capture the trace.

Code instrumentation is one of the most common techniques to register the behavior of programs and to measure its performance [14, 16, 17]. Nevertheless, due to the impact of instrumentation on the systems' performance [41], this type of profiling is not usually performed on production environments. Other works [18, 21, 44] use sampling to reduce the cost of instrumentation. However, since these profilers proceed by statistical analysis, they may lead to wrong performance measures. Gprof [23] is an example of a profiler that uses both instrumentation and sampling. Instrumentation is used to gather caller information and the actual timing values are obtained by sampling. A work that discusses the advantages and disadvantages of these techniques is presented by Hall [24].

3.2 What Should We Measure?

In general, developers want to have as much information as they can get, and in more detail as possible, so that developers can effectively change the web application when necessary, assuring its success and performance, increasing its life time. However, our approach is to increase developers awareness on simple performance information rather than providing complex and expensive to get information.

To understand what would help developers to focus in the most relevant areas, both in terms of efficiency and relevance, we prepared an inquiry with possibly relevant metrics that we presented to the most experienced group of *OutSystems* developers. The idea was to understand which metrics would help the most these developers to anticipate inefficiency. The following metrics were included on the inquiry:

1. Execution count and average execution time for actions, screens and queries.
2. Unstable code markers (elements that raise unexpected errors more than X% of times).
3. Caller frequency (who calls an action or screen?).
4. For each web screen - page size, average session size, average view state size.
5. Average of number of records returned for each query.
6. Hit ratio for cache mechanisms.
7. Code coverage (code "used" in last X weeks).
8. Common user navigation path.
9. Bounce rate (percentage of visitors that hit the website on a given page and then leave it without visiting any other pages).
10. Exit rate (percentage of visitors that leave the system on a given page).

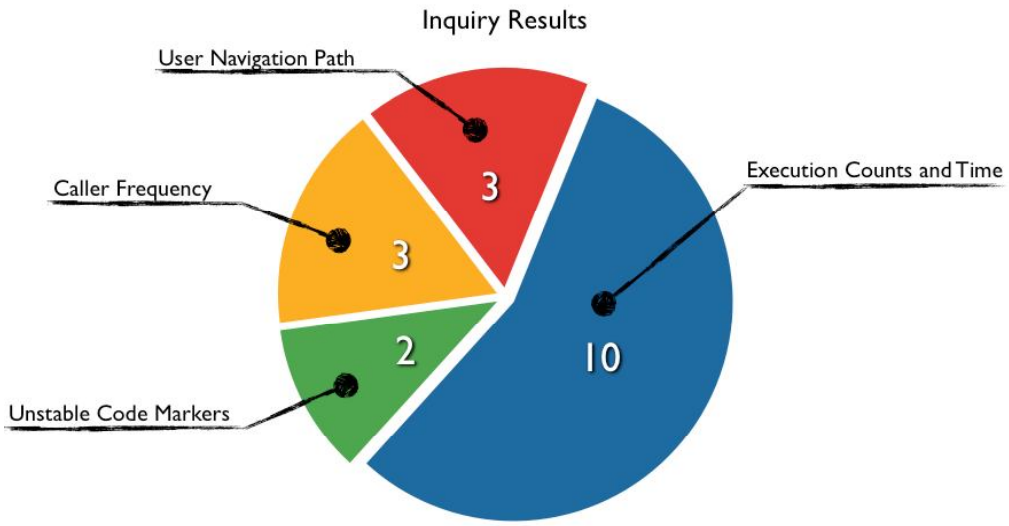

Figure 3.1: Inquiry results

Each developer could select only two metrics. As depicted in *Figure* 3.1, all developers inquired, considered more relevant to measure the execution counts and time of the programming elements. Developers also considered relevant to measure the calls frequency and to be informed about the common user navigation path, both metrics were selected four times. Finally, two developers considered relevant to know which elements raise unexpected errors. The other metrics were not selected. Note here that this study was done to guide the initial metrics to capture.

The results of the survey were clear. Developers considered more relevant to have execution counts and average execution time for actions, queries and screen preparations (special kind of action) and to keep the impact of the data collection as low as possible. This means that it is absolutely necessary to measure as few properties as possible.

The execution counts and average execution time of actions, queries and screen preparations, allow developers to understand what is heavier on a web application, either by being executed more often or by taking more time to execute. Showing this information to developers results in two clear benefits:

1. It decreases the risk of code modifications having unexpected and significant impact on performance. Developers know they are changing sensible code and will correctly identify the risk of changing it.

2. Developers can monitor and identify bottlenecks of the applications before reaching critical status for end-users.

In the next chapter we describe some techniques and methodologies that were studied as a basis for the development and implementation phase of our project.

4

Profiling Techniques

4.1 Profiling of Web Applications

With the growth of the internet and the increase demand of software, in particular of web applications, sometimes companies are forced to shorten development software time and to release software without performing enough analysis and testing.

In the context of web applications, techniques have been studied to validate these applications before being deployed into production [22, 28, 29, 39]. A web application is considered valid when it is guaranteed that all paths in the site which satisfy a selected criterion are properly exercised before delivery.

After an application has been implemented, it usually goes through a phase of testing before going into production, this phase is performed in a development environment. Techniques and tools to test and profile software on a development environment [2, 3, 8, 20], have also been created and studied. These techniques and tools try to address the best practices to improve the software development team productivity and software quality.

The techniques and tools used to analyze these applications, in development environments, usually collect huge amounts of data. There are visualization techniques that can be very effective, transforming program-execution data into visual information [15, 37, 38].

4.1.1 Gprof

Gprof [23] is a classical profiler, widely referred among the community. This profiler was created to help developers figuring which functions should be optimized, either because they are called very often or because they take significant time to run.

Gprof is a call graph profiler that provides information about which functions in a program call which other functions; about the number of calls; and about the amount of time spent in each one. Therefore it gathers three pieces of information during program execution: call counts and execution times for each profiled routine, and the arcs of the dynamic call graph traversed by this execution of the program. By post-processing of this data, it builds the dynamic call graph for the execution of the program and propagate times along the edges of the graph to attribute times for routines to the routines that invoke them.

To avoid impact on the running of the program it gathers profiling data in memory during program execution and condense it to a file as the profiled program exits. This file is then processed by a separate program to produce the listing of the profile data. An advantage of this approach is that the profile data for several executions of a program can be combined by the post-processing to provide a profile of many executions.

Gprof uses instrumentation to gather caller information and the actual timing values are obtained by sampling.

When a program is compiled with gprof support, monitoring routines are inserted at strategic points in the code to produce a trace of events. To measure the execution time it uses a method that samples the value of the program counter at some interval, and infers execution time from the distribution of the samples within the program.

4.1.2 Google Analytics

Several tools for profiling web applications have been created and are highly popular, but most of these are useful for marketers and managers. They usually provide metrics about traffic of data on the network and marketing effectiveness by following users paths. This kind of profilers can determine, for instance, the direct impact of a specific marketing campaign by analyzing the entry point of each user.

Google Analytics [4, 27] is perhaps the most significative example inside this category - its a solution that analyzes traffic data, helps clients to better target their ads, to strength marketing initiatives, and to create web applications that better match their market goals.

The web analytics provided by *Google*, can help, for instance, to determine whether blog visitors have a positive impact on a web application, or which visitors acquisition channels work best and to what extent these should be increased or decreased.

The purpose of *Google Analytics* is to give the knowledge from which, marketeers and managers, can make informed decisions about changing online strategies.

In this solution, all data collection, processing and maintenance are managed by *Google*. When a user interacts with a web application with the *Google Analytics Tracking Code* (GATC), visitor data as page URL, unique ID, screen resolution, are collected. For each pageview, the

GATC sends information to *Google* data collection servers. Finally, reports are typically displayed (updated) every 4 hours.

4.2 Measures

Conclusions of *Section* 3.2 pointed to measure the execution counts and the execution time of screens, actions and queries. Thus we analyze *Frequency* and *Time*.

Frequency is the measure that describes the number of occurrences of a repeating event per time unit. For instance:

- "How many times did users select the button "send to a friend" of the page newProfile.html last week?"
- "How many times was the page bestProfile.html rendered during last week?"
- "How many times is a function called?"
- "How many times is a function or a block of code executed?"

The other property is a component of the measurement system and is used to compare the duration of events and the intervals between them. It adds information to the questions:

- "What was the average execution time of the function X during last week?"
- "Which functions are taking more than 6 seconds of execution time?"
- "What is taking longer since the new version was deployed?"

Execution counts and average execution time are two metrics that make sense to show together. They allow, for instance, to tackle the following situation where a development team have access to the metrics:

- "last week the average execution time of function X was 6 seconds"
- "last week the average execution time of function Y was 1 second"

A proactive team would probably try to optimize function X instead of function Y. However, if function X was executed 10 times in that period in opposition to 10000 times of function Y, our optimization on function X would have almost no impact on the end-users experience. A careful optimization on function Y would certainly have more impact.

4.3 Execution Time

There are several techniques to measure execution time which are characterized by four key attributes [41]:

- **Accuracy** - Defines how far is the measured time from the actual execution time of a procedure. When a measurement is made, there is usually some amount of error, the measurement is usually a result of *actual execution time +/- some amount of error*, where *some amount of error* corresponds to the *accuracy*.

- **Difficulty** - Defines the necessary effort to obtain measurements. A method that only requires the user to run the code and it produces an answer, is considered easy. A method that requires usage of a logic analyzer and filtering of data to obtain answers is considered hard.

- **Granularity** - Defines the size of the part of the code being measured. For example, coarse granularity methods would generally measure execution time per process, per procedure or per function basis. A method with fine granularity can measure execution time of a loop or even a single instruction.

- **Resolution** - Represents the measure limitation. For example, a stop watch measures with a 0.01 sec resolution, while a logic analyzer might be able to measure with a resolution of 50 nsec.

The design of the software can also have a major impact on the ability to obtain measurements of execution time. If a software has a single entry and exit point for any part of it that needs to be measured, and those points are designed consistently for all code segments that have similar functionality then achieving accurate metrics can be possible.

Software Analyzer method is a term used for software tools designed for measuring execution time as CodeTest [3]. Usually software analyzers are based on the system clock and thus the resolution is on the order of a millisecond. A good analyzer not only provides information about functions and processes, but also means to measure execution time of loops, blocks of code and statements. For example, in our context, timing trace must be correlated with the *OutSystems* DSL elements to identify which element is responsible for each period of execution.

Each programming language provides a mechanism to retrieve the current time from the system. These mechanisms save the system time on specific instants and then compute the time intervals by subtracting the values of the system taken at different moments. The Java, for example, creates an object that can be used as a stopwatch to measure the time execution of code blocks.

4.4 Nonblocking Counters

The traditional way to coordinate access to shared variables is to use blocking algorithms, ensuring that all access to shared fields is done holding the appropriate lock.

Synchronization assures that whichever thread holds the lock, will have exclusive access to those variables, and changes to those variables will become visible to other threads only when they acquire the lock.

The counter in *Listing* 4.1 is thread-safe. To safely increment the counter, the thread take the current value, add one to it, and write the new value out, all as a single operation that cannot be interrupted by another thread. Otherwise, if two threads tried to execute the increment simultaneously, an unlucky interleaving of operations would result in the counter being incremented only once, instead of twice.

When multiple threads ask for the same lock at the same time, one acquires the block and the others block. JVMs typically increment blocking by suspending the blocked thread and rescheduling it later. This implementation can cause a significant delay relative to the few instructions protected by the lock. More details about this example can be consulted in [6].

Listing 4.1: Example of a blocking algorithm to increment a counter

```
public final class Counter {
   private long value = 0;

   public synchronized long getValue() {
      return value;
   }

   public synchronized long increment() {
   return ++value;
   }
}
```

But there are alternatives, for example, the compare-and-set (CAS) method. The CAS method basically works by trying to update the value of the counter, but it fails if some other thread changed the value since it was looked.

The CAS method includes a memory location (M), the expected old value (A), and a new value (B). It starts by reading a value A from an address M, then it performs a computation to derive a new value B, and then use CAS to change the value of V from A to B. The CAS succeeds if the value at V has not been changed in the meantime. If another thread did modify the variable, the CAS would detect it (and fail) and the algorithm could retry the operation.

A simple nonblocking algorithm using CAS, can have a performance advantage over the lock-based versions. Loosing threads can retry immediately rather than being suspended and rescheduled. And even with few failed CAS operations, this approach seems to be faster than being rescheduled [33]. *Listing* 4.2 shows the counter class rewritten to use CAS instead of locking. More details about this example can be consulted in a document written by Brian Goetz [5].

In this chapter we described some methodologies and techniques for profiling. We started by referring some work related with profiling of web applications and by describing the classical *Gprof*. We also presented *Google Analytics* which, although is a known web profiling analyzer, it is not directly related with our work. We want to apply a classic profiling strategy to measure other kind of properties. We want to implement profiling techniques that retrieve relevant metrics for developers and IT managers.

We also described some techniques that we will use on our solution. For instance, we will use instrumentation to collect runtime performance data. We are also interested in measuring time and frequency and providing these two metrics together.

Listing 4.2: Example of a Counter class with CAS

```
public class CasCounter {
   private Simulated value;

   public int getValue() {
      return value.getValue();
   }

   public void increment() {
      int oldValue = value.getValue();
      while (value.compareAndSwap(oldValue, oldValue + 1) != oldValue)
         oldValue = value.getValue();
   }
}
```

5

Profiling of Web Applications in OutSystems

In this chapter, we describe the most relevant aspects of the work developed during this dissertation. We start by addressing the compilation process of the *OutSystems* language and then, we describe the design, implementation and development of our profiling system to run in real world *OutSystems* environments.

We focus first on the instrumentation of the *OutSystems* compiler. And then, we describe how our system collects, aggregates, and transports data from the production environment to the development environment, and how they are made available to the developer, crossing different architectural layers of the *Agile Platform* avoiding significant impact.

5.1 Code Generation

When a web application is created with *Service Studio* it can generate Microsoft .NET [36] or JAVA [42] code, depending on the target web application server, either IIS [34] or JBOSS [7]. In this work, we defined changes in the generation of code .NET.

The DSL compiler generate C# [25, 26] for the applicational logic and ASP [35] for the creation of web pages.

The syntactic analysis and validation of code is done in *Service Studio*. So, the *OutSystems* DSL compiler receives a model that was previously validated and then proceeds with its transcription to .NET.

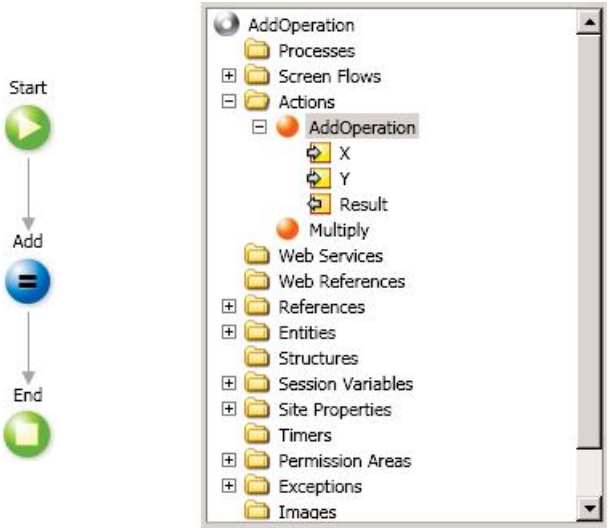

Figure 5.1: Action and *eSpace* tree example

5.2 Instrumenting Code

In our work we use code instrumentation because we are interested, in the first place, in giving a flow of continuous real-data information to developers, and probabilistic results (sampling profilers) are not as accurate as desired. Another and perhaps the most important one is that the impact of the data collection is diminished by associating counters to high level programming elements.

We collect data by instrumentation of the code generated by the DSL compiler. Remember that a developer invokes a process (*1-Click Publish*) in *Service Studio* to deploy the application to a development environment. At that moment, *Service Studio* send the *eSpace* to the compiler that generates a standard J2EE or .NET application. Then, when the application is ready, it is transported to a production environment where it is used by final end-users.

We adapted the compiler to insert probes in the generated code for each specific element we chose to monitor. These probes are used every time that one of these elements are executed.

When an application is deployed to a development environment it is automatically instrumented. And when it is transported to a production environment, end-users interact with the application and these probes are reached and data is recorded and stored in a structured way.

In *Figure* 5.1 we can see a simple example of an action and of an *eSpace* tree. The purpose of action *Add* is to sum the value of variable X with the value of variable Y. In *Listing* 5.1 we can see the generated code for that action before the profiler was implemented. In *Listing* 5.2 we can see the generated code for the same action with the profiling system running. After the method declaration, we declare and initialize an object *Stopwatch*. The object *Stopwatch* has two important methods: a *Start* method that stores the time at the moment it was called, and a *Stop* method that calculate the difference between the time at the moment the method *Start* and *Stop* were called. We also declare a boolean variable *_profilerError*, and we initialize the variable as *false*.

In the end of the method, if the code rose an expected error, we change the value of *_profileError* to *true*. Then we call the method *Stop* of the object *Stopwatch* and we finally call the method *ProfileElement* that stores all this information in a data structure that is integrated in the running environment of the web application. Note here that, as we will see in the next section, every web application is hosted by a running environment that contains a data structure with the counters for all elements of the application being monitored. When probes are reached, they interact with these data structures according to the usage of the applications.

In this example we show the instrumented code for an action flow, but our profiling system is measuring the execution time, execution counts and error counts for other elements of the language: buttons, queries, actions and iterator cycles (for each). The profiler also measures the average number of iterations of *for each* cycles.

Listing 5.1: (BEFORE) Example of generated code for an action

```
public static void ActionAddOperation(HeContext heContext, int inParamX,
int inParamY, out int outParamResult) {
   lcoAddOperation result = new lcoAddOperation();
   lcvAddOperation localVars = new lcvAddOperation(inParamX, inParamY);
   try {
      //Add
      result.outParamResult = (localVars.inParamX +
      localVars.inParamY; //Result = X + Y
   } //try

   finally{
      outParamResult = result.outParamResult;
   }
}
```

5.3 Collecting Runtime Performance Data

In order to minimize the impact of profiling on applications' performance, we adopt an architecture with multiple layers and priorities to collect, aggregate, and transport data from the running application back to the developer. Profiling code inlined in the application code to count the number of times an action gets executed or the time it takes to terminate is crucial and runs with the highest priority, and hence must be designed to have minimum impact on execution time. Collected data is stored close to the programming elements being monitored. This data must then be aggregated and transported across the *Platform Server* architecture. This is performed in persistent state, with less and less priority and more and more spread in time. We next explain in greater detail each step of the process.

Each running application (an *eSpace*) is hosted by a running environment which holds a data structure containing a counter for the executions, errors and the total execution time for all its programming elements being monitored. In order to transport the data through the different architecture layers of the *Agile Platform* we then use services that run on low priority to avoid competing with the processing of web applications.

Listing 5.2: (AFTER) Example of generated code for an action

```
public static void ActionAddOperation(HeContext heContext, int inParamX,
int inParamY, out int outParamResult) {
   System.Diagnostics.Stopwatch _profilerStopWatch = new System Diagnostics.Stopwatch();
   bool _profilerError = false;
   try {
      lcoAddOperation result = new lcoAddOperation();
      lcvAddOperation localVars = new lcvAddOperation(inParamX, inParamY);

      try {
         //Add
         result.outParamResult = (localVars.inParamX +
         localVars.inParamY; //Result = X + Y
      } //try

      finally{
         outParamResult = result.outParamResult;
      }
   catch (Exception) {
      _profilerError = true;
      throw;
   }

   finally {
      _profilerStopwatch.Stop();
      OutSystems.HubEdition.RuntimePlatform.Profiler.ProfileElement("H6ekx40c6k+vc5_QWMKTQA",
      "/UserActions.NrVoz+h8XUCKuWhvZPp7yw", _profilerStopWatch.ElapsedMilliseconds,
      _profilerError);
      }
   }
}
```

Figure 5.2 represents the data structures of applications that are running in a *Front-end Server*. For the sake of simplicity, the image is simplified. Although each data structure only shows the execution counts for some actions, these data structures contain the metrics for all elements being monitored. The probes that were inserted by the DSL compiler, during the generation of the code for the applications, interact with these data structures according to their usage. For example, when a user runs the application Enterprise Manager and proceed with the user login, the execution counts for the action login is incremented.

Since the application server recycles running applications in regular time intervals we store data into persistent state whenever necessary. Approximately every 15 minutes, all measurements in the running environment are pushed to the nearest *Log Service* and stored in the database and all counters and timers are put to zeros.

The next step is to store data in a secondary persistent state, the filesystem. This intermediate step of storing results in the *Log Service* could be by-passed, but the *Log Service* is optimized to avoid the impact of logging information on the application's performance, this service runs with low priority in the *Front-end Server* so it never competes with the normal processing of web applications.

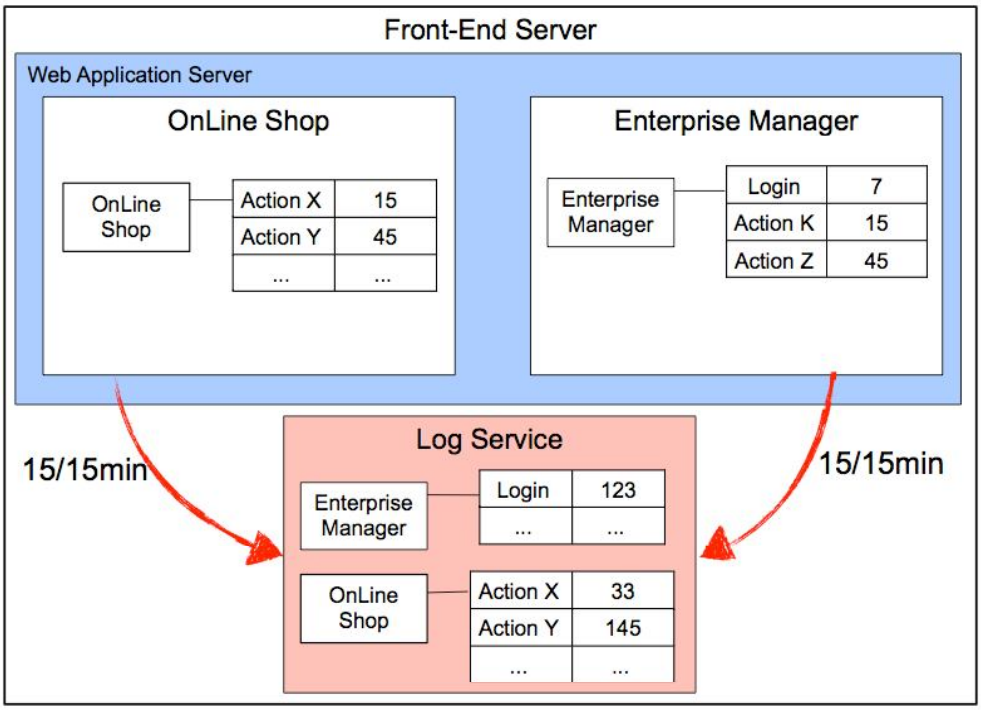

Figure 5.2: Example of data structures to store metrics

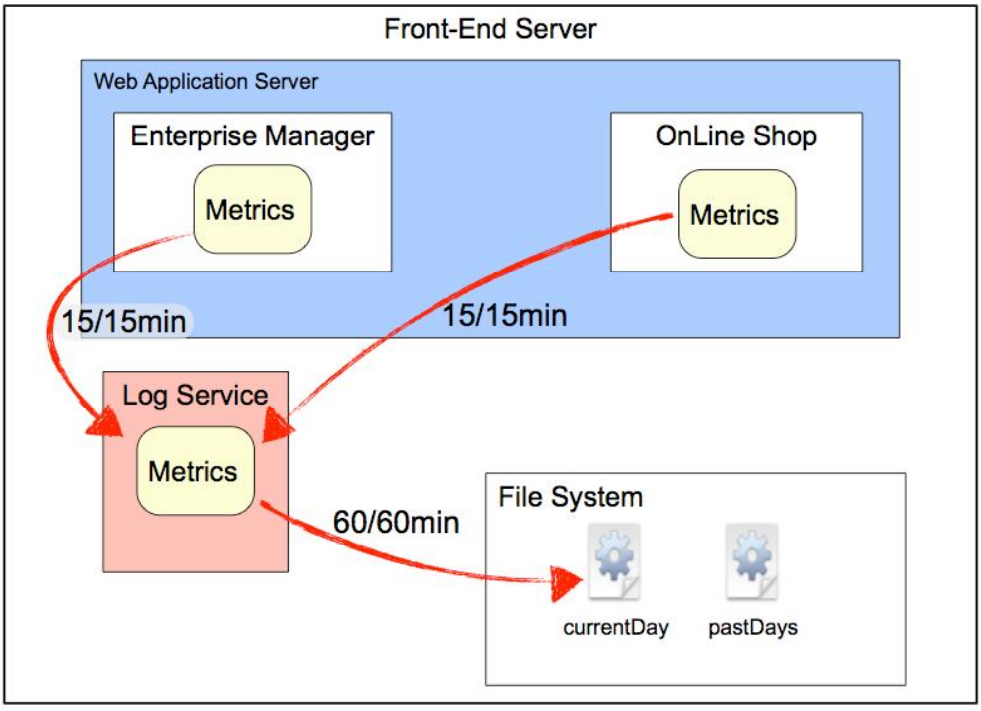

Figure 5.3: Front-end data flow

Figure 5.3 shows the data flow inside a *Front-end Server*. Every 60 minutes, the *Log Service* appends the *currentDay* file with the collected data. Note here that the file system contains two files, a *currentDay* file that contains the metrics of the current day, and a *pastDays* file that maintain a pool of the metrics for the last 7 days, where the oldest is replaced by a new one each day. Hence, the profiling system has a continuous flow of information of about one week.

These intervals of time arise from a balance between the risk of loosing data and the efficiency of storing that can be tuned according to the experience using the system. For example, if the *web application server*, for some reason, stops working, we just lose, in the worst case, metrics of about 15 minutes. The *Log Service* is much more stable, but, even so, if for some reason it stops working, in the worst case, with very low probability, we lose metrics of about 60 minutes. In addition, the amount of collected data in 60 minutes doesn't make the process of aggregation and storage inefficient.

5.4 Log Service and Data Structures

A *Front-end Server* can host more than one web application and, in every *Front-end Server* there is a *Log Service* that contains all metrics for the elements being monitored of the web applications. To handle these metrics we designed the data structures that are depicted in *Figure* 5.4 and are described as follows (note that each *Log Service* contains only runtime performance data for the *Front-end Server* in which it is running):

- A *Log Service* contains a *ProfilerWeekData* that contains a list of objects of the type *ProfilerData* with the metrics of the last 7 days, and a object of *ProfilerData* with the metrics of the current day.

- The object *ProfilerData* contains a *date* and a *HashTable* where a key is a *eSpaceKey* and a value is an object of the type *eSpaceProfilerData*.

- An *EspaceProfilerData* contains a *HashTable* where a key is *elementKey* and a value is an object of the type *elementProfilerData*.

- Finally, an *ElementProfilerData* contains all the metrics of an element: the total time execution, the number of executions, the number of errors, and the number of iterations if the element being monitored is a *for each*.

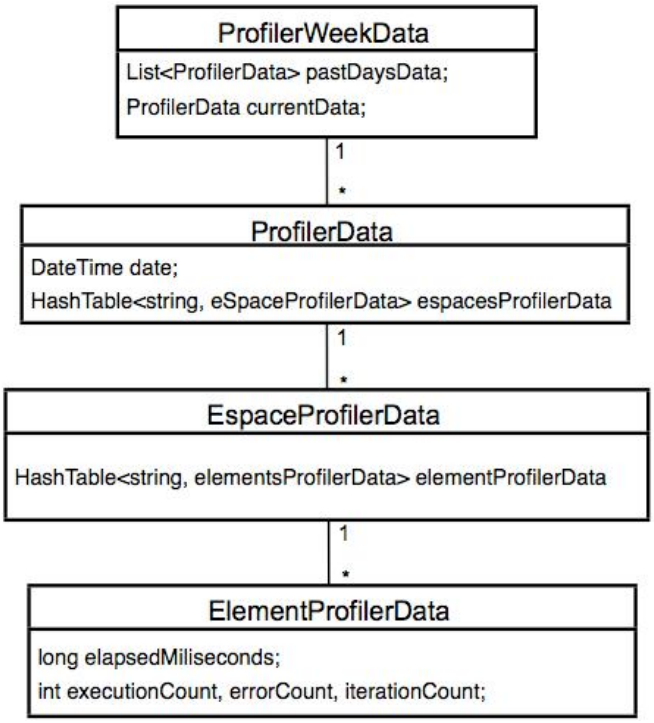

Figure 5.4: Data structures

We now describe the activity of a *Log Service* in our profiling system:

- On *Start* it gets all the metrics that are stored in the file system.
- It maintains all the metrics of the web applications running in the given *Front-end Server*.
- It aggregates and receives metrics from the web applications that are running in the same *Front-end Server*. Every web application, in a regular interval of 15 minutes send its metrics to the nearest *Log Service*.
- Every 60 minutes, it stores the metrics of the current day (*currentData*) in a secondary persistent state, the file system.
- Every 24 hours:
  - it adds the metrics of the current day (*currentData*) to the list with the metrics of the last 7 days (*pastDaysData*) replacing the oldest one;
  - creates a new *currentData* for the new day;
  - stores the metrics of the last 7 days (*pastDaysData*) and replace the old file *currentDay* by a new one.
- On *Stop* it stores all the metrics in the file system.

5.5 Transporting Metrics to Service Studio

Our goal is to provide production metrics inside *Service Studio*. In *Figure* 5.5 we can see both the development and production environments, this figure also represents the situation when there is no performance data in the development database for a given *eSpace*. This process follows the steps:

1. When a developer opens an *eSpace* with *Service Studio*, a request for performance data for the given *eSpace*, is sent to the development *Service Center*.

2. Since there is no fresh data available in the development database, the development *Service Center* sends a request for metrics, for the given *eSpace*, to the production *Service Center*.

3. Production *Service Center* communicates with every *Front-end Server* configured, by contacting the *Log Service* and requesting the performance collected data.

4. Each *Log Service* sends back to the production *Service Center* all the metrics for the given *eSpace*. The *Service Center* is responsible for aggregating the metrics received from the different *Front-end Servers*.

5. Production *Service Center* sends back to the development *Service Center* all the metrics.

6. Finally, development *Service Center* provides the metrics to *Service Studio* and caches all performance data in the development database.

Figure 5.6 represents the situation when the performance data for a given *eSpace* is available in the development database. This process follows the steps:

1. When a developer opens an *eSpace* with *Service Studio*, a request for performance data for the given *eSpace* is sent to the development *Service Center*.

2. Since the performance data for the given *eSpace* is available in the development database, the development *Service Center* retrieves it.

3. Finally, all the performance data for the given *eSpace* is provided to *Service Studio*.

Note that this process is asynchronous. The developer can work on *Service Studio* while the data is being imported. The transport of data is done on demand to avoid unnecessary communication and process between the two environments. For example, if a development team stop the development process during a period of time, it isn't necessary to transport the runtime performance data of the web application to the development environment.

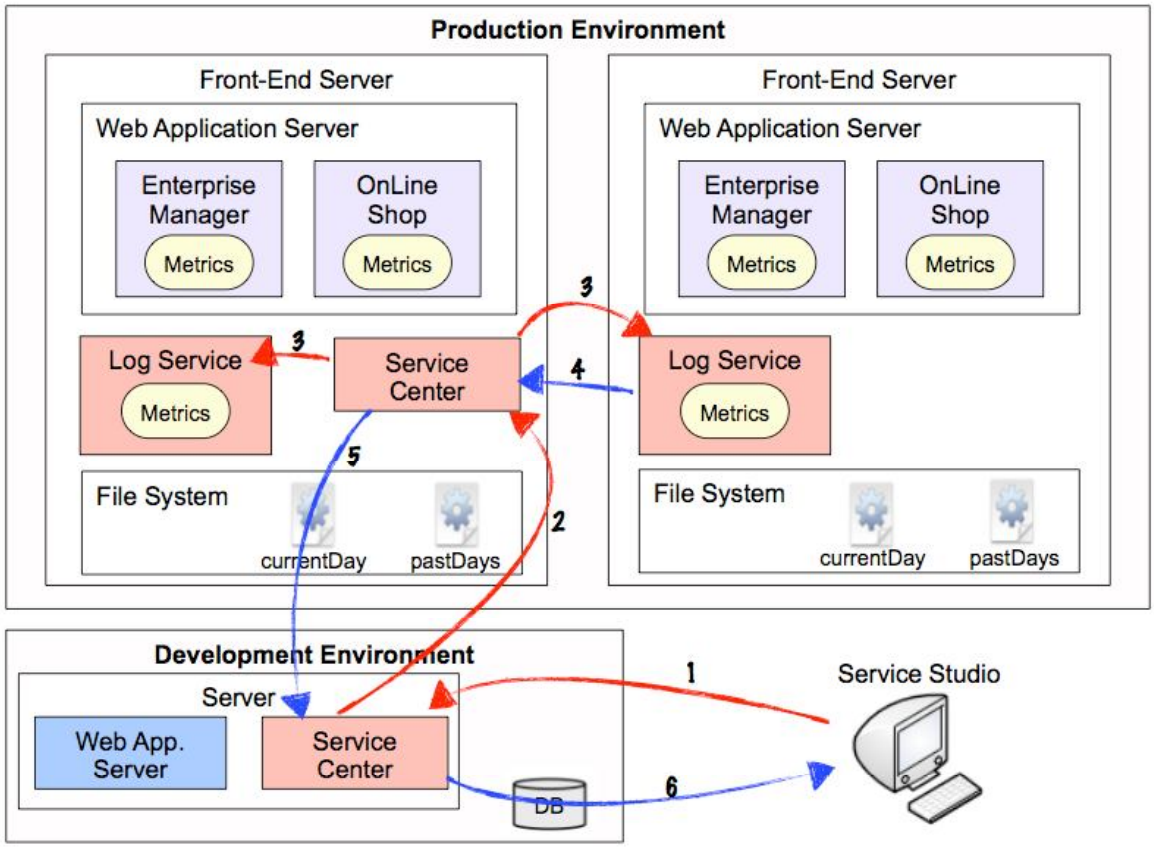

Figure 5.5: Data flow when there is no performance data cached

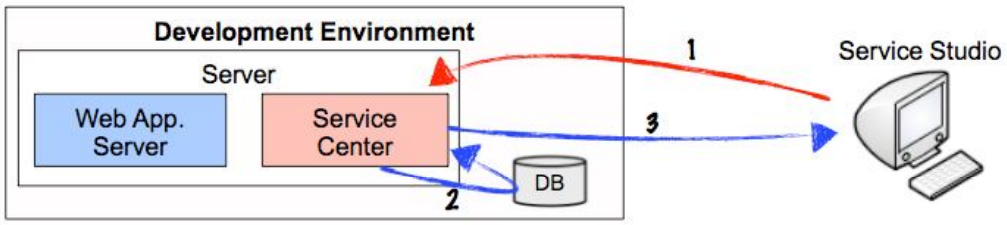

Figure 5.6: Data flow when performance data is cached in the development environment

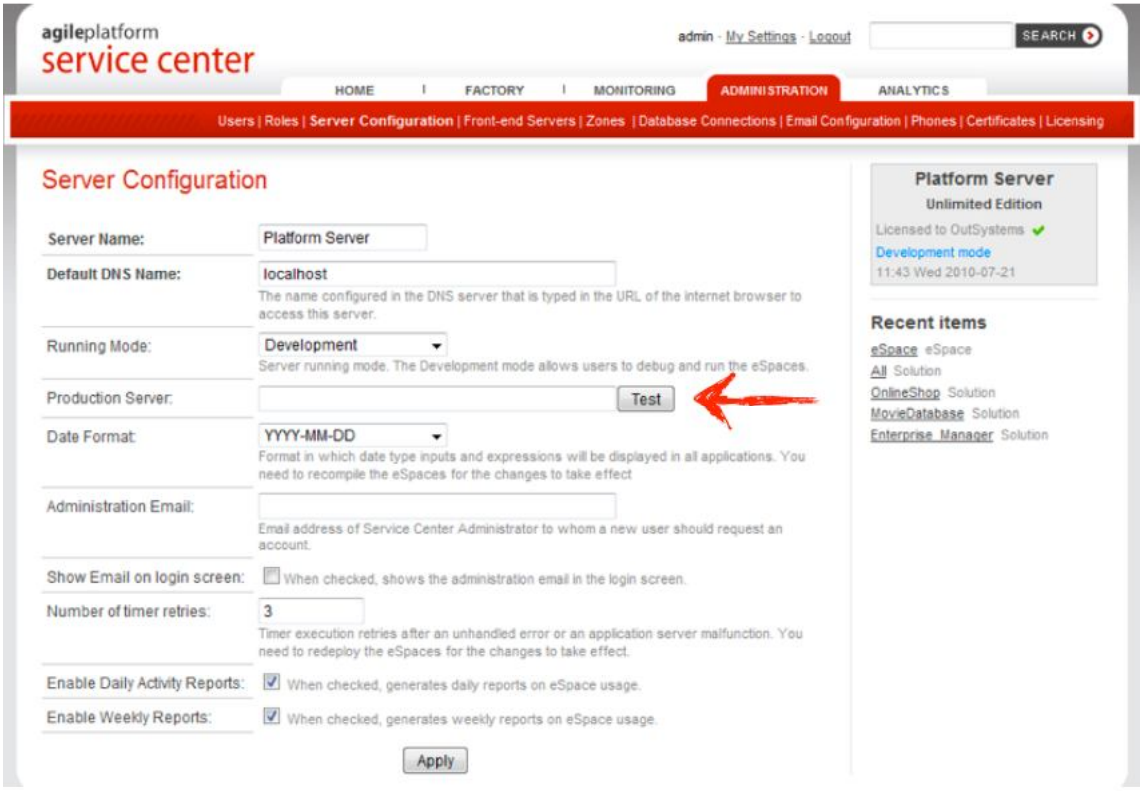

Figure 5.7: API for defining the address of production *Service Center*

5.6 Connecting Development and Production Environments

As we described in the previous section, the data is transported from the production to the development environment through communications initiated from the development *Service Center* to the production *Service Center*.

To implement our profiling system, we connected the development and production environments. Note that, until now, these two environments were completely disconnected. There wasn't any communication protocol between these two environments.

Production *Service Center* exposes a web service that is consumed by the development *Service Center*. To establish this communication, a person that has administration privileges, can define the production *Service Center* address in the development *Service Center*. *Figure* 5.7 shows the new interface where this parameter is defined.

The security challenge of this connection was not addressed since we are not transporting business data but statistical data which is only useful for those that can access the application model. However, it will be probably solved in a future work by establishing a trust relationship between the development *Service Center* and the production *Service Center*.

5.7 Decorating Service Studio

In the beginning of this document we listed the challenges of this work and one of them was to decorate *Service Studio* with profiling information without cluttering the existing environment and without significant impact to its performance.

To provide information in *Service Studio*, after inquiring the *OutSystems* user interface team, we decided to provide the metrics for the DSL elements on the *properties pane*. The *Properties Pane* is where the developer can see and define the properties of the selected element, either in the *Flow Canvas* or in the *eSpace Tree*. *Figure* 5.8 and *Figure* 5.9 show the metrics that are provided when a developer either selects a *Preparation Screen* or an *Action* in the *eSpace Tree*.

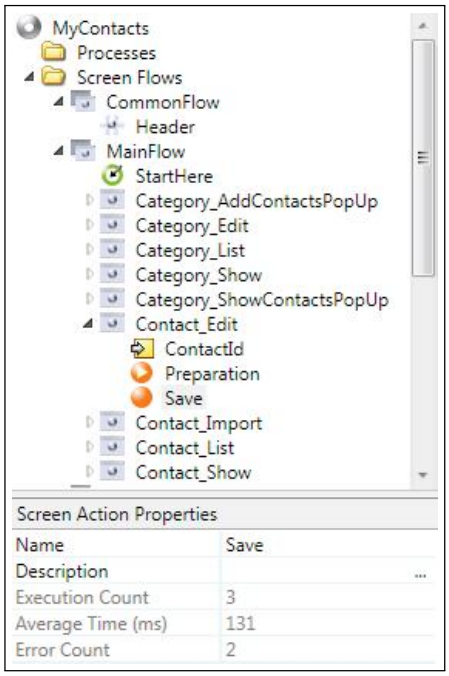

Figure 5.8: Example of the *properties pane* for an *action*

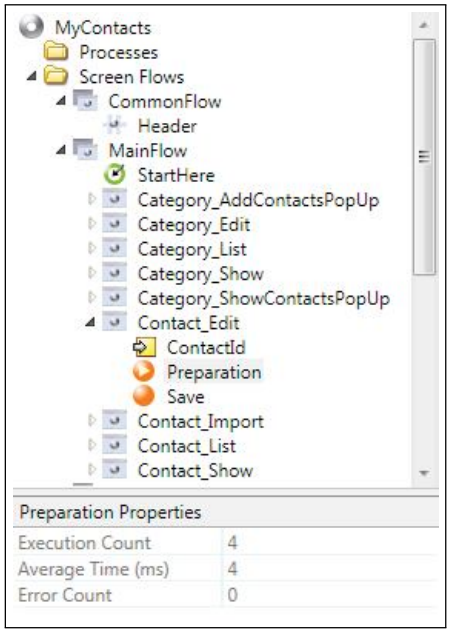

Figure 5.9: Example of the *properties pane* for a *preparation screen*

When a developer either selects an *Action* or a *Preparation Screen* in the *eSpace Tree*, the *Action Flow* opens in the *Flow Canvas*. For the sake of simplicity, we show in *Figure* 5.10 a simple example of an *Action Flow*. We can see the *Action Flow* of the *Preparation Screen* for the page *Contact_List*. Since the query *GetContacts* is selected, we can see in the properties pane, the metrics of the last week for that element. In this case, the action is simple, but as we will see in the next section *Action Flows* can be very complex and that's where our profiling system become more valuable. When an *Action Flow* has more elements, a developer can navigate along it, by selecting the different language elements and having access to their performance metrics.

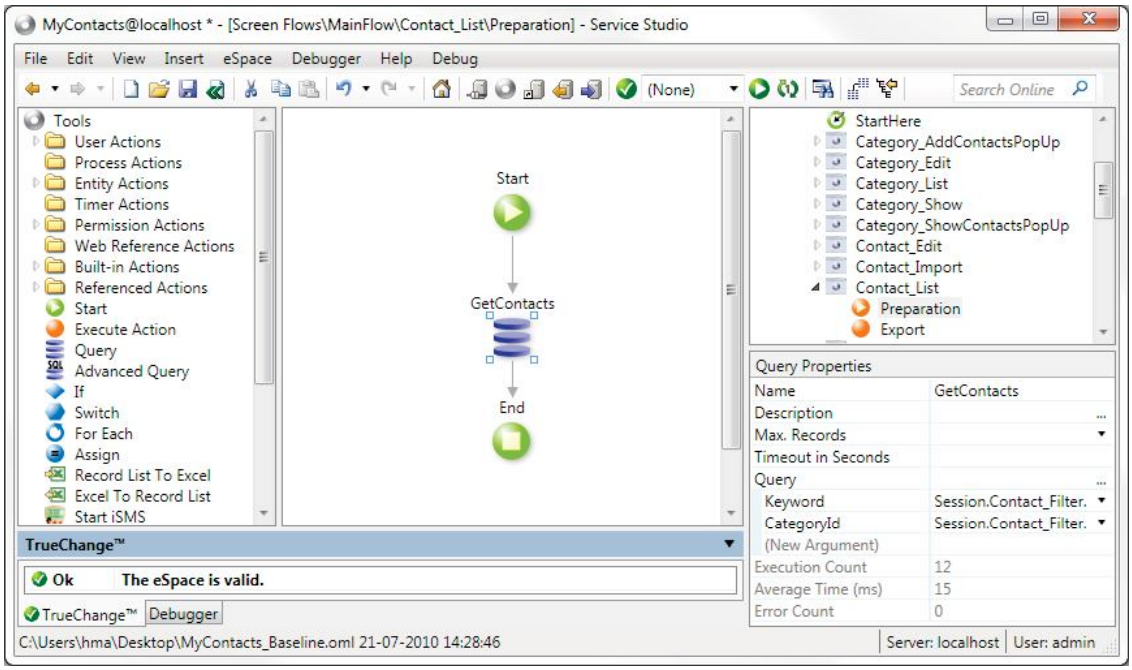

Figure 5.10: Example of the *Properties Pane* when a query is selected in a *Action Flow*

To help identifying bottlenecks or stress points in the web applications, we added a warning sign that appears in the right bottom corner of the language elements that either rose unexpected errors more than 25% of the times they were executed or have an average execution time of more than 200 milliseconds. Again these values are guessed to be satisfiable but should be tuned with the help of real experience. Figure 5.11 shows an action that has an advanced query that is decorated with the warning sign, meaning that this query is either taking more than 200 milliseconds to be executed or, more than 25% of the times it is executed, it raises unexpected errors.

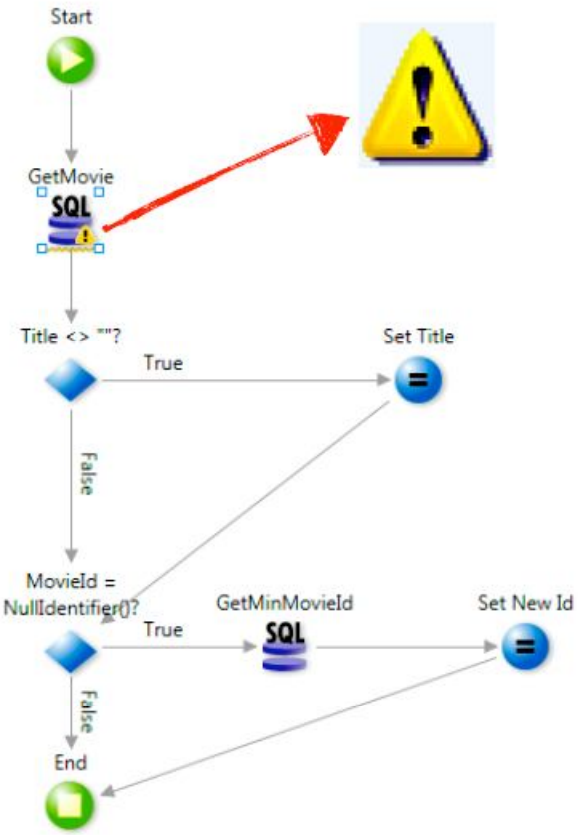

Figure 5.11: Example of an advanced query decorated with a waning sign

6

Results Analysis

In this chapter, we analyze the results of our profiling system. In the first part we focus the benefits of having production metrics inside *Service Studio*. For instance, we describe a real example of a developer experience while looking for a performance bottleneck. We then present a summary of the results of performance tests that were made to validate and analyze the impact in production of our profiling system.

We used WAPT [12], a load and stress testing tool that provides a consistent way of testing web applications and web servers. WAPT uses a number of techniques to simulate real load conditions. It creates a simulation of many different users coming from different IP addresses, each with their own parameters: cookies, input data for various page forms, name, connection speed and their own specific path through the site.

Our approach was to analyze the performance characteristics of a web application and of a web server, under various load conditions. This metrics were recorded with the profiling system running, and compared with the metrics recorded with the profiling system *off*.

6.1 Developer Experience while Looking for a Bottleneck

By having early access to real performance metrics, developers easily detect stress points in applications, allowing for timely tuning actions, before reaching critical conditions for end-users.

However, there are cases where development teams stop the development process during a period of time and a web application may reach critical conditions. In these cases, based on the developers opinion, our profiling system decreases dramatically the time to detect the bottleneck.

Metrics also help developers assessing the impact of changing intensively used parts of existing applications. This is relevant, because uninformed modifications may have significant impact on the applications performance.

Finally, our profiling system, promotes an agile approach to scalability requirements. Since the developer focus only the main problems. Developers just optimize what is really necessary to increase the performance of the application.

Example

Now, we describe a hands on example that was applied to a real scenario. Consider that a web application is deployed to a production environment. After a period of time, critical feedback about the application is given back to a developer. The problem is related with the time response of the application. When a user opens a specific page, the new screen takes too much time to render.

To solve this performance issue, the developer needs to open the application *eSpace*, selects the *Screen Preparation* for the page which takes too much time to render, find which logic is responsible for this delay, and finaly proceed with an optimization. Remember that a *Screen Preparation* is a type of an *Action Flow* that always runs before a specific screen is rendered.

Until now, this is done in a trial and error iterative process, that requires developers' experience and some speculation about where the time is effectively being spent in the production environment (as the development environment has a different context in the database). And the developer needed to capture the metrics using either a small debug session or explicit audit operations in the middle of the code. In a small debug session, the developer tries to understand where the time is being spent. This is a experience statistically irrelevant. In the other hand, with explicit audit operations, the developer needs to collect metrics before and after every element of the language. This is a hard work and usually requires a considerably long period of time.

With our profiling system the developer has to open the *Preparation Screen* that is related to the page defined. And the metrics for the last week of that screen are provided. As we can see in *Figure* 6.1, the average execution time of the screen is 573 milliseconds and it was executed 114 times. The developer can now navigate on the flow, by selecting the different language nodes and check the metrics of each element. Note that this is a complex *Action Flow* and these figures just show part of it. As shown in *Figure* 6.1, the element responsible for most of the time execution of the *Screen Preparation*, is the action *Solution_GetReferences*.

Knowing this, the developer opens the action *Solution_GetReferences* to open the related *Action Flow* and navigates along it. In this case, as we can see in *Figure* 6.2, the *forEach* is responsible for almost all of the average time execution of the action *Solution_GetReferences*. Since the bottleneck was detected, it can now be resolved.

If this process would been done without this system, the developer would have to design an ad hoc scheme to collect metrics either by using audit events or a debugging session. The application would probably have to run for another period of time or to be tested and analyzed in a controlled development environment. However, this collected data is not representative of the real world production environment, as it is not possible to reproduce the size of data nor the number of different users. The current processes to identify bottlenecks is complex and requires time.

In this real example we reduced the time for detecting the bottleneck from hours or even days to just a few minutes.

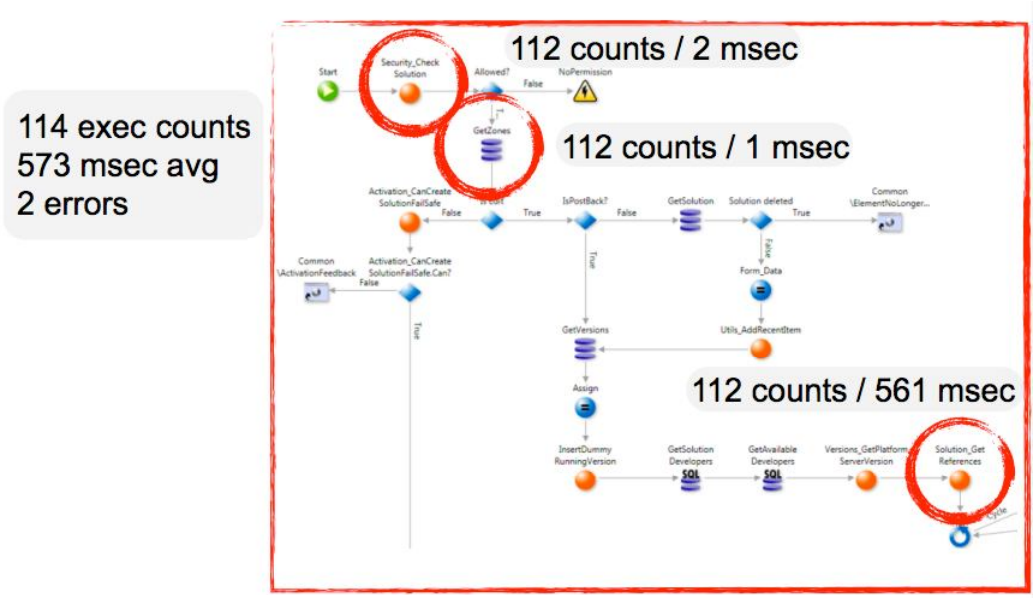

Figure 6.1: Example: detecting the bottleneck in a *prepraration screen*

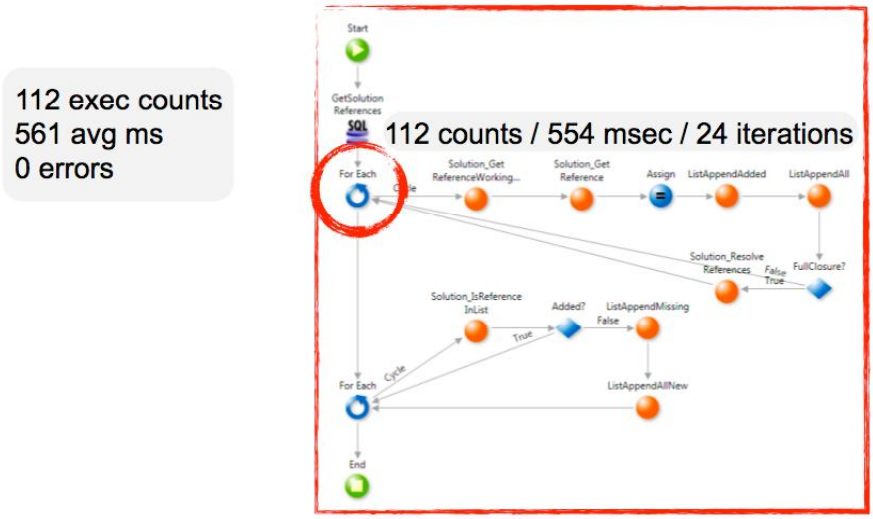

Figure 6.2: Example: detecting the bottleneck in *action flow*

6.2 Disk Space Usage

As described in *Chapter* 5, in every *Front-end Server* configured in a farm, the profiler stores the metrics in the file system for persistency purposes.

To measure the disk usage of the profiling system in each *Front-end Server* we tested a web application that has 60 screens, which is considered a medium size web application in *OutSystems* context. With WAPT, we simulated 300 users accessing the web application visiting all web application screens. Note here, that only some parameters have impact on the size of the files containing the runtime performance metrics: the number of elements being monitored that are typically correlated with the number of screens (every screen has more or less the same number of elements being monitored); the number of days; and the number of web applications.

After our test, the disk usage of the profiling system in the *Front-end Server* was of 144KB per day and per application. We cannot be exact about the disk usage of our profiling system in a real world environment but we can estimate it. For that, we multiply this value for 7, since our profiling system has a continuous flow of information of 7 days, and we then multiply the result for 40. A *Front-end Server* can host 40 web applications, sometimes even more. We are considering that all web applications are similar to this one, containing 60 different screens. To sum up, we estimate that our profiling system has a total and permanent disk usage of approximately 40MB (144KB x 7 x 40 = 39.375 MB), which is acceptable in terms of disk space.

6.3 Runtime Performance Impact

To see if our profiling system is having impact on the responsiveness performance of web applications, we did several load tests. We analyzed the *average response time* of a web application and the number of pages that are executed per second (*pages per second*), and we compared the results with the profiling system running and with the profiling system inactive. The number of *pages per second* is a valuable result of testing an application capacity and overall performance. The metric *average response time* is also an important characteristic of load testing an application, it measures web user experience. Response time graph tells how long a user waits for server response to his request.

First, we did several tests, increasing the number of users accessing a web application, and passing through all possible screens. Our goal in this first phase, was to see the maximum *pages per second* that the web application was capable to execute while running in a sustained way. In both cases, the web application, couldn't execute more that an average of 55 *pages per second*. We reached this value, in both cases, on tests with 100 users. We did several tests, with more users, as 150, 200, 300 and we conclude that the web application in the testing environment could not execute more than an average of 55 *pages per second*. This result was the same with the profiling running, and with the profiling inactive. In figures 6.3, 6.4, 6.5, 6.6, 6.7 and 6.8, we can see that the average of executed *pages per second* is always near the value of 55. The small triangles in the graph represent the number of users that are concurrently accessing the web application.

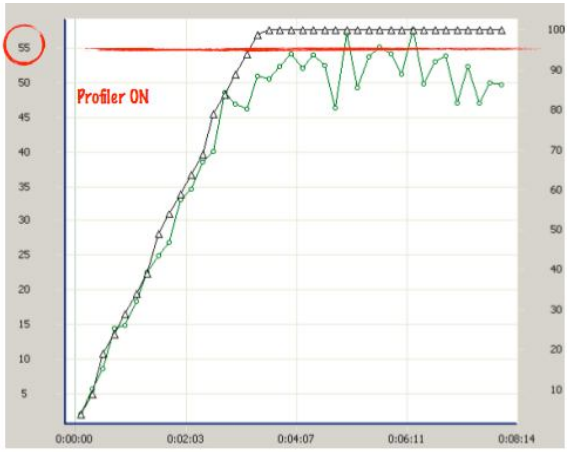

Figure 6.3: Graph of *pages per second*: test with 100 users, profiling ON

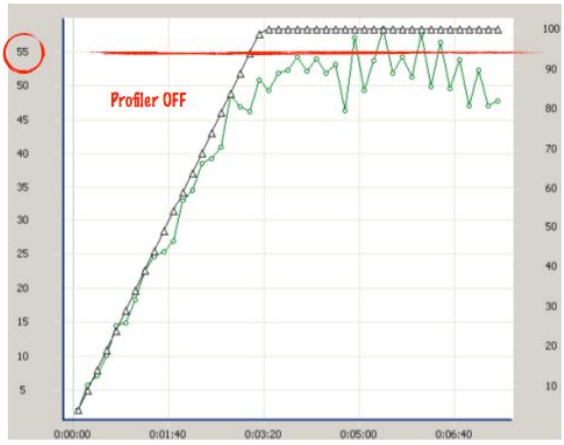

Figure 6.4: Graph of *pages per second*: test with 100 users, profiling OFF

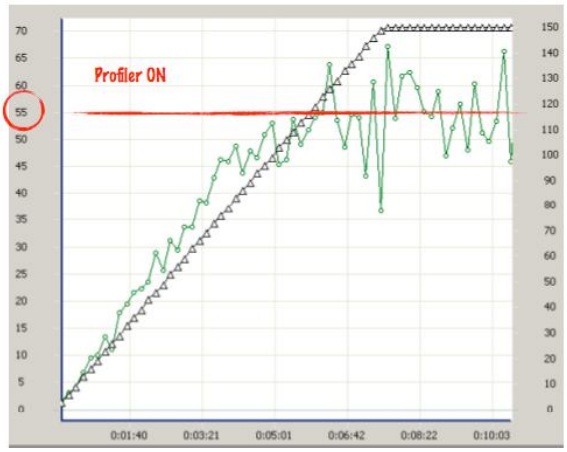

Figure 6.5: Graph of *pages per second*: test with 150 users, profiling ON

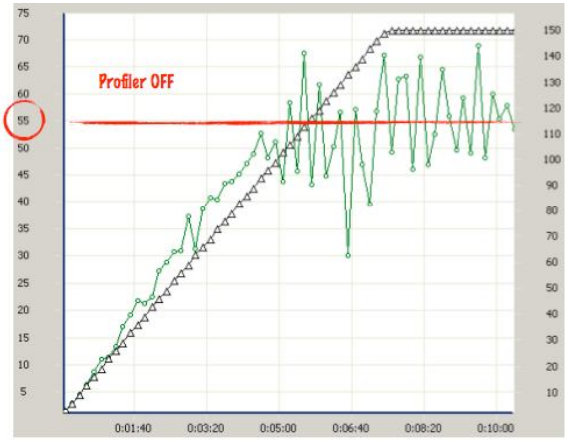

Figure 6.6: Graph of *pages per second*: test with 150 users, profiling OFF

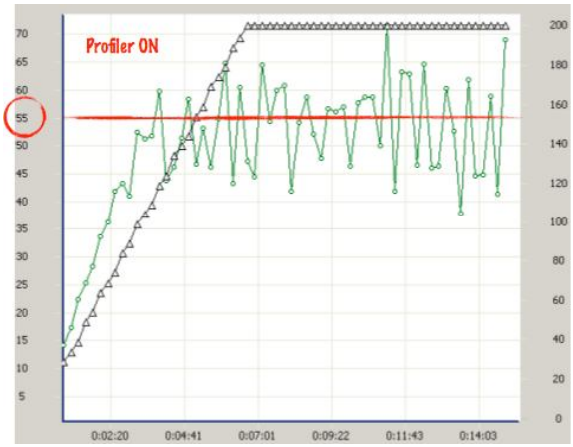

Figure 6.7: Graph of *pages per second*: test with 200 users, profiling ON

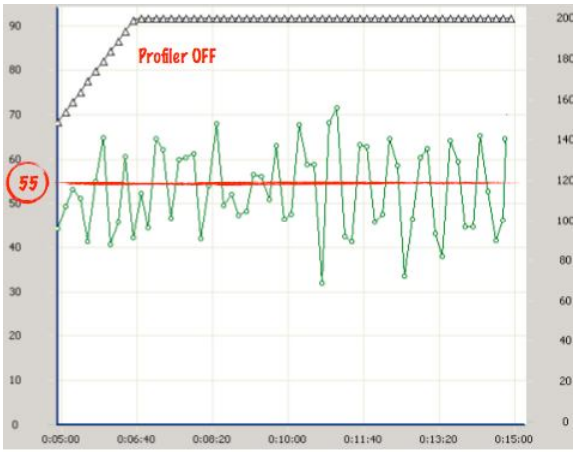

Figure 6.8: Graph of *pages per second*: test with 200 users, profiling OFF

The next step, was to analyze the graph of *average time response*. Note that in the previous tests, the *average time response* is not represented. But, since the web application could just execute 55 pages per second, as long as we increase the number of users, the response time also increases. Our approach, in this second phase, was to identity the load conditions to reach a specific value of *average response time*. We did several tests, increasing the number of users until the performance of the web application started to break. We considered that a web application started to break when the value of the *average response time* reached the value of 0.5 seconds. *Figure* 6.9 and *Figure* 6.10 show that comparing to the graph with the profiling *off*, the average response time is longer with the profiling *on*.

Note that in Figure 6.9, although the test ends up with 150 users accessing the web application, the graph reaches the 0.5 seconds when the web application is being used by 120 users. But in *Figure* 6.10, the graph of the *average response time* only reaches the value of 0.5 seconds with 150 users.

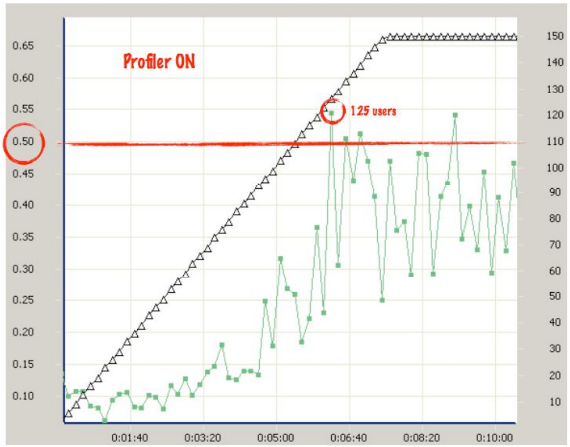

Figure 6.9: Graph of *average response time*: test with 150 users, profiling ON

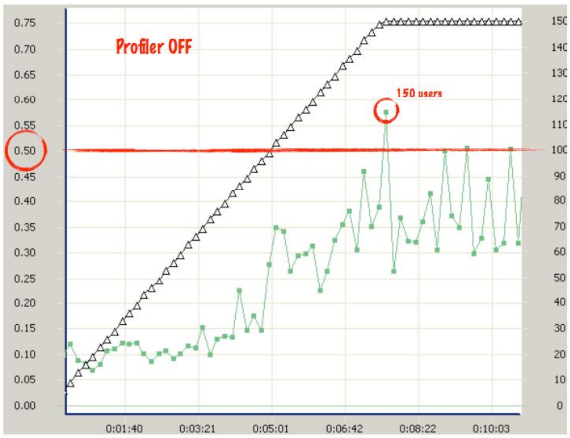

Figure 6.10: Graph of *average response time*: test with 150 users, profiling OFF

6.4 Concluding Remarks

In this chapter we first referred to the value of this our profiling system, describing some benefits of providing production runtime performance information to developers and we described a real example of a developer experience looking for a performance bottleneck. We then analyzed the impact of our profiling in production systems and we conclude that it keeps the interference level and the performance impact in production environments at an acceptable level.

7

Conclusions

7.1 Work Experience

This thesis is integrated in the Research and Development (R&D) team of the *OutSystems* company.

During the first phase of this thesis, we described a preliminary study and a possible solution for a profiling system. Some profiling techniques were studied as a basis for the decisions that we later made along the development and implementation phase.

It was necessary to understand the functionalities and purposes of the different components of the *Agile Platform*: *Service Studio*, *Integration Studio*, *Service Center* and *Platform Server*. It was also important to understand the DSL compiler and the *OutSystems* language. The next step was to start skimming the *Agile Platform* which is formed by a set of more than 70 projects developed in *Microsoft Visual Studio 2008* with a code base of more than 1 million lines. It was important to focus the DSL compiler to understand the process of code generation.

The first phase occupied 40% of the time available and was already made in collaboration with the company.

The development and implementation phase was addressed on a full time basis at the company, and it was integrated in the team responsible for the development of the version 6.0 of the *Agile Platform*. In this phase, it followed the *OutSystems Agile Methodology*, a methodology based on SCRUM Agile Methodologies [30, 40], for control and organization of projects.

The development was done in iterative process, that suffered some changes. For example, the architecture of the profiling system was first designed to use the database production, but later, with the help of the engineering team we considered an advantage to avoid bulk inserts on the database and use the file system as a secondary persistent state. With the arise of new challenges, we were forced to reinforce the solution and rethink our decisions along time. Therefore, it was extremely important to be integrated in a team of specialists that provided insightful comments and relevant discussions.

In the context of the project developed during this master thesis, it was written and submitted a paper, titled *Profiling of Real-World Web Aplications* [13], to the International Symposium on Software Testing and Analysis, Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging, where it was later presented (13th July in Trento, Italy). The paper was accepted by the conference committee and is available in the ACM Digital Library. This was a joint work with Lucio Ferrão from *OutSystems* and with João Costa Seco from the CITI research center of *Universidade Nova de Lisboa*.

The final result of this project is fully functional and integrated in the development branch of the *Agile Platform*. It is now waiting for a product management release date decision, with the corresponding deeper investment in quality control, usability, and product marketing teams.

7.2 Conclusions

Non-functional requirements of enterprise web applications, like performance, are usually assessed and analyzed in simulated environments before being released into production. However, it is not easy to simulate a real-world environment and the effective use of the system, leading to poor and expensive performance data collection. Moreover, in agile methodologies, where development is focused in the fast time to market and getting early feedback from end-users, upfront estimation and forward thinking about scalability is not in the top priorities. This constrains even more performance analysis and tests, as developers are only aware of performance issues when critical feedback from production systems is given back to development. This commonly leads to enterprise web applications with scalability problems, and low responsiveness. All this results in bad end-user experience and high maintenance costs.

We were primarily motivated to answer two different questions with our work:

- Can we collect real world data about performance of web applications without significant impact in the end-user experience?
- Can we give feedback to developers in a way that impact analysis is improved and that anticipation of performance issues is achieved?

Profiling of web applications is not a common task in industrial environments. There are some explanations for that fact among which we find the heterogeneous context of web applications the most relevant. Collecting data in all tiers of an application and gathering it in a meaningful way is not a trivial task. A second reason for this is related with the performance degradation caused by instrumentation and data collection.

In this work, we presented an architecture of a profiling system that collects real world data from web applications. By focusing on a DSL we reduced the impact of measurements on the performance and the end-user experience. By giving feedback to developers proactively we improved the whole agile development cycle. Although this is not a common practice, it should be, since the value is obvious. The problem is that most development environments fail to offer the integration and abstraction level that exists in *OutSystems* context.

Finally, we believe that we achieved positive results since the goals that were considered to make this a successful project were accomplished:

1. To collect metrics at runtime, without significant degradation on the server and applications performance.

2. To gather and transport data from a production environment to a development environment.

3. To decorate *Service Studio* with profiling information, without cluttering the existing environment and without significant impact to its performance.

4. To achieve positive results on the agile maintenance process of enterprise web applications. This goal was subdivided in: helping developers to easily detect stress points and bottlenecks; allowing developers to fix them before reaching critical status for end-users; and increasing the developers' awareness when changing intensively used code.

7.3 Future Work

In order to completely prove the value of our study, we will have to analyze and measure the benefits of the profiling system along time. In the future, we need to have statistics on how these runtime performance metrics contribute to increase the end-user satisfaction, and to decrease the costs of maintenance.

The work developed during this master thesis may also evolve to provide other types of metrics. One important metric is measuring how many cache hits occur in an application. This would help developers validating the configurations of cache. For example, if a query is cached during 2 minutes, and usually has 20 cache hits during this period, the developer can conclude that the use of cache in this element is increasing the execution performance since, approximately every 2 minutes, it is avoiding 19 accesses to the database. This metric gives to developers, a possibility to balance the parameters of cache to increase its value. But there are other type of metrics that can be interesting, as the data tables size, for example to understand if a menu should be auto-complete instead of drop-down (for a database with considerably data, it is erroneous to use a drop-down menu) or the most frequent navigation path, to increase the awareness about the hot spot of the application.

This work may also contribute to the development of future works, for instance:

- Using statistical information of real data flow from production environments, it is possible to generate specially optimized code for the most relevant data flows. Without such information, it is only possible to optimize the generated code for the worst scenario;
- Using the information of changes between production and quality assurance environments, together with the profiling information, it is possible to answer two relevant questions: a) did I test all the statistically relevant code that has changed since last quality assurance tests? b) did I test all the code that is frequently used in production?
- Using the profiler information it is possible to suggest relevant unit tests in development environments;
- Using the profiler information it is possible to offer real-time impact analysis warnings of changes in the database model and changes in the long term process models.

Appendix

A.1 Glossary

- **1-Click Publishing** - operation that involves the following steps: Save (save the eSpace in a specified folder), Upload (upload it to the Platform Server), Compile (the oml file is translated in the Platform Server into .NET or JAVA and Deploy (operation that updates the area that contains the last published version.

- **Action Flow** - a guided graph, potentially cyclical, that contains an initial node and a set of terminal nodes. For example, the developer can have a flow to control elements (If, For Each, ..., End) and can invoke other actions. Its the graph that represents the Screen Action.

- **Bottleneck** - a bottleneck is a phenomenon where the performance or capacity of a system is limited by a single or limited number of components or resources. In programming it can be a loop, a method or other block of code depending on the granularity used to refer the bottleneck point.

- **Domain Specific Language (DSL)** - is a specification language, dedicated to a particular problem domain, a particular problem representation technique, and/or a particular solution technique.

- **Development Environment** - the environment in which developers design, create, modify and test applications.

- **Deployment Controller Server** - is in charge of compiling *eSpaces*, and deploying the compilation result in the *Front-end Servers*. There is only one *Deployment Controller Server* for each *Platform Server* installation.
- **Integration Studio** - visual environment to create connectors, actions, entities, structures in the *OutSystems* terminology, for integrating with existing enterprise systems.
- **Environment** - an environment represents some stage of the application development. In this document we refer to two types of environments: *development environment* and *production environment*. Both contain a web server, service center, a compiler, a database, and the running web applications.
- **eSpace** - a web application project. We can develop and change *eSpaces* in *Service Studio*. They contain all definition needed for developing and managing web applications, such as web pages, business logic, database tables, security settings, and lot of more.
- **OML** - atands for *OutSystems Markup Language* and is the format by which the *eSpaces* are saved to disk. It is also the file extension (.oml) for the *eSpaces*.
- **Refactoring** - code refactoring is the process of changing a computer program's internal structure without modifying its external functional behavior or existing functionality, in order to improve internal quality attributes.
- **Service Center** - a web console that enables the operational management of the Agile Platform. It provides, for example, version control and configuration management of all web business applications, services, and other resources.
- **Service Studio** - the visual development environment to fully develop projects: design, create, modify and finally test them. With Service Studio developers assemble all components necessary to completely define a web business application, without writing any code. The tool enables the modeling of user interfaces, business logic, web services, scheduled processes and security.
- **Screen Preparation** - runs before the screen is rendered. Can be used to get data o display on the screen.
- **Screen Action** - runs on specific events on the screen (usually the click of a button).
- **Platform Server** - runtime platform that controls all runtime, deployment, and management activities for every application designed with the Service Studio in a standard application server infrastructure.
- **Production Environment** - an web application is inside a Production Environment when is on a Server and available on Web.

Bibliography

[1] Index of Mathematicians, Dec 2009. http://turnbull.mcs.st-and.ac.uk/history/Mathematicians/Thomson.html.

[2] Bullseye. A code coverage analyser, Jan 2010. http://www.bullseye.com/.

[3] CodeTEST Software Analysis Tools, Jan 2010. http://www.freescale.com.

[4] Google Analytics, Jan 2010. http://www.google.com/analytics/.

[5] Java theory and practice: Going atomic, Jan 2010. https://www.ibm.com.

[6] Java theory and practice: Introduction to nonblocking algorithms, Jan 2010. http://www.ibm.com.

[7] JBOSS, Jul 2010. http://jboss.org.

[8] JTest. Java Static Analysis, Code Review, Unit Testing, Runtime Error Detection, Jan 2010. http://www.parasoft.com.

[9] Netcraft.com. Web site statistics 2010, June 2010. http://news.netcraft.com.

[10] OutSystems, Jan 2010. http://www.OutSystems.com/.

[11] Service Studio Help, OutSystems Agile Platform 5.1, Jun 2010. http://www.outsystems.com/help/servicestudio/5.1.

[12] Web Application Testing - WAPT, Jul 2010. http://www.loadtestingtool.com/.

[13] Hugo Menino Aguiar, João Costa Seco, and Lúcio Ferrão. Profiling of real-world web applications. In *ISSTA' 10: Proceedings of the 2010 International Symposium on Software Testing & Analysis, and co-Located Workshops MIT'10, PADTAD'10, STOV'10, and WODA'10*, Trento, Italy, 2010. ACM.

[14] Matthew Arnold and Barbara G. Ryder. A framework for reducing the cost of instrumented code. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 168–179, New York, NY, USA, 2001. ACM.

[15] Marla J. Baker and Stephen G. Eick. Space-filling software visualization. In *Readings in information visualization: using vision to think*, pages 160–182, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[16] Thomas Ball and James R. Larus. Optimally profiling and tracing programs. In *ACM Trans. Program. Lang. Syst.*, volume 16, pages 1319–1360, New York, NY, USA, 1994. ACM.

[17] Geoffrey Owen Blandy, Maher Afif Saba, and Robert J. Urquhart. Code instrumentation system with non intrusive means and cache memory optimization for dynamic monitoring of code segments. Number 5940618, August 1999.

[18] M. Burrows, U. Erlingsson, S.-T. A. Leung, M. T. Vandevoorde, C. A. Waldspurger, K. Walker, and W. E. Weihl. Efficient and flexible value sampling. *SIGPLAN Not.*, 35(11):160–167, 2000.

[19] Dhyani Devanshu, Keong Ng Wee, and Bhowmick S. Sourav. A survey of web metrics. *ACM Computing Surveys (CSUR)*, 34(4):269–503, 2002.

[20] Elizabeth D. Dolan and Jorge J. Moré. Benchmarking optimization software with performance profiles. *ArXiv Computer Science e-prints*, cs.MS/0102001, 2001.

[21] Evelyn Duesterwald and Vasanth Bala. Software profiling for hot path prediction: less is more. In *ASPLOS-IX: Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, pages 202–211, New York, NY, USA, 2000. ACM.

[22] Sebastian Elbaum, Gregg Rothermel, Srikanth Karre, and Marc Fisher II. Leveraging user-session data to support web application testing. *IEEE Trans. Softw. Eng.*, 31(3):187–202, 2005.

[23] Susan L. Graham, Marshall Kessler, and K. McKusick. Gprof: a call graph execution profiler. *ACM SIGPLAN Notices*, 17(6):120–126, 1982.

[24] Robert J. Hall. Call path profiling. *ACM, Proceedings of the 14th international conference on Software Engineering*, pages 296–306, 1992.

[25] Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. *C# Language Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[26] Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. *C# Programming Language, The (2nd Edition) (Microsoft .Net Deveolpment Series)*. Addison-Wesley Professional, June 2006.

[27] Jerri Ledford and Mary E. Tyler. *Google Analytics 2.0*. Wiley Publishing, 2007.

[28] Chien-Hung Liu, David C. Kung, Pei Hsia, and Chih-Tung Hsu. Structural testing of web applications. In *ISSRE '00: Proceedings of the 11th International Symposium on Software Reliability Engineering*, page 84, Washington, DC, USA, 2000. IEEE Computer Society.

[29] G. Di Lucca, A. Fasolino, and F. Faralli. Testing web applications. In *ICSM '02: Proceedings of the International Conference on Software Maintenance (ICSM'02)*, page 310, Washington, DC, USA, 2002. IEEE Computer Society.

[30] Robert C. Martin. *Agile Software Development, Principles, Patterns, and Practices*. Alan Apt Series. Prentice Hall, Upper Saddle River, NJ, October 2002.

[31] Marjen Mernik, Jan Heering, and Anthony M. Sloan. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316–344, 2005.

[32] Edu Metz and Raimondas Lencevicius. Efficient instrumentation for performance profiling, 2003.

[33] Maged M. Michael and Michael L. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *J. Parallel Distrib. Comput.*, 51(1):1–26, 1998.

[34] Microsoft. Microsoft IIS, Jul 2010. http://www.iis.net/.

[35] Microsoft. MSDN ASP, Jul 2010. msdn.microsoft.com.

[36] Microsoft. MSDN .NET Development Website, Jul 2010. http://msdn.microsoft.com.

[37] Alessandro Orso, James Jones, and Mary Jean Harrold. Visualization of program-execution data for deployed software. In *SoftVis '03: Proceedings of the 2003 ACM symposium on Software visualization*, pages 67–ff, New York, NY, USA, 2003. ACM.

[38] Steven P. Reiss and Manos Renieris. Encoding program executions. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 221–230, Washington, DC, USA, 2001. IEEE Computer Society.

[39] Filippo Ricca and Paolo Tonella. Analysis and testing of web applications. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 25–34, Washington, DC, USA, 2001. IEEE Computer Society.

[40] Ken Schwaber and Mike Beedle. *Agile Software Development with Scrum*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[41] David B. Stewart. Measuring execution time and real-time performance. In *In: Proceedings of the Embedded Systems Conference (ESC SF)*, pages 1–15, 2002.

[42] Sun. Java 2EE, Jul 2010. java.sun.com/javaee/.

[43] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *ACM SIGPLAN Notices*, 35:26–36, 2000.

[44] John Whaley. A portable sampling-based profiler for java virtual machines. In *JAVA '00: Proceedings of the ACM 2000 conference on Java Grande*, pages 78–87, New York, NY, USA, 2000. ACM.